LVRL: Reducing the Gap between Immersive VR and Desktop Graphical Applications

Daniel Trindade, Lucas Teixeira, Manuel Loaiza,
Felipe Carvalho, Alberto Raposo, Ismael Santos

Pontifical Catholic University of Rio de Janeiro, Brazil

Guest Editors: Luciano P. Soares, Liliane S. Machado

ABSTRACT **—The emergence of cheaper technologies for immersive environments has considerably increased the interest in Virtual Reality applications. However, VR frameworks currently available force user applications to be developed specifically for them. This increases the cost of converting an existing application to another virtual reality environment. This paper proposes a new framework, the LVRL (*Lightweight Virtual Reality Libraries*), which allows the creation or conversion of existing applications to VR without changing the application's structure. The LVRL's main objective is to provide a non-intrusive and transparent programming interface that allows the development of VR applications by non-VR developers. This paper describes LVRL's architecture, features, usage, and benefits obtained by applications using it.**

1. Introduction

The evolution of human-computer interaction has led to the development of different interfaces and interaction mechanisms. Many of these new interfaces are not yet mature, and issues related with the clear definition of an application's context and technological requirements are still under investigation. Most of the tasks performed in a conventional desktop computer are related to text editing, file organization, use of tables and mathematical calculations, etc. However, other areas of application have emerged, e.g. 2D image editing, animation, Computer Aided Design (CAD), interactive 3D visualization, entertainment, and games. The inherent characteristics of these classes of applications have created a demand which was not fully met by the desktop metaphor as it was originally proposed to represent an office through WIMP interfaces.

The emergence of three-dimensional interactive applications showed the limitation of conventional 2D desktop devices (mouse, keyboard, and monitor). Such a conventional setup does not meet all the interaction needs related to the additional dimension in this new environment. The main evidence supporting this fact is the existence of a research line focused exclusively on 3D computer graphics that is as old as the very development of WIMP interfaces. The early efforts were guided by the evolution of scientific visualization and flight simulator applications driving the development of a new set of interactive hardware, including head mounted displays, data gloves, and CAVES, as well as new interaction techniques for 3D environments.

The use of virtual reality (VR) as a tool for regular users and developers is increasing. For a long time, two major obstacles have increased this gap, the first one related to the high prices of equipments, and the second one related to the lack of standardization of devices and interaction. The high cost of equipments is being gradually overcome by new technologies that made it possible for many centers to have access to immersive systems such as Caves and PowerWalls. On the other hand, the diversity of interaction devices and 3D interaction techniques, as well as the lack of standard definitions, are still problems for the development of an immersive VR application.

One key issue in immersive VR development is the huge amount of time wasted on implementing infrastructure aspects like devices/interaction management, output synchronization, and rendering distribution, leaving little time for implementation of innovative solutions. The subject of toolkits and innovation is widely exploited by Greenberg in [13], and is of fundamental importance for the development of this research. According to Greenberg, for a new area to develop it is necessary to give everyday programmers the ability to test their creativity by means of programming tools that remove low level implementation burdens.

To achieve these goals within a project with complex requirements, such as those of VR, there is a great need for reuse of code. At this point a toolkit is extremely useful because it encapsulates several functionalities and allows the developer to focus on what is really desired.

There are some toolkits available for the development of immersive VR applications, such as VrJuggler [7], Blender-Cave [12], Eon Studio [20], Avango NG [16], 3DVIA Virtools [5] and INVRS [6]. However, the majority of these toolkits are coupled to a specific rendering system, making it difficult to adapt an already functional desktop graphical application to an immersive VR system. As a consequence, applications often need to be completely rewritten for this kind of environment. This burden is also an impediment to creative development in VR.

However, all this work is not actually necessary since the code needed for the creation of an immersive VR application may be developed in a way to be used in existing applications. Roughly speaking, an immersive VR application differs from a conventional desktop graphical application in two main aspects: i) it should support multiple video outputs with different points of view, and ii) the user cannot use conventional interaction devices, such as mouse and keyboard, since that user interacts standing in front of the immersive environment (Figure 1). In this paper we propose a non-intrusive framework to enable the conversion of graphical desktop applications into immersive ones, aiming to provide programmers that are not experts in VR the ability to produce immersive applications. We also perform usability tests that show that the interaction provided by the framework is capable of being used by users of engineering visualization applications.

In section 2 we present the common VR tools used today and the reasons that hinder the conversion of 3D software from desktop to an immersive environment approach. Then in section 3 we explain each of the framework's libraries and also show how it's usage makes the transition from desktop setup to immersive setup seamless. In section 4 we show six applications which are used as study cases to the application of LVRL. In section 5 we discuss the characteristics and contributions of the framework, and we present results of usability tests related to the interaction techniques provided by LVLR. Finally, some conclusions and future work are presented in section 6.

2. Related Work

The existing frameworks for VR often try to support the following functionalities: rendering multiple views, device management for 3D interaction, and rendering distribution over a LAN. As mentioned above, while attempting to support these features, they also direct their development process to a particular platform.

BlenderCave [12], *Virtools* [5], and *EON Studio* [20] are authoring tools for the development of interactive 3D applications. They have rendering systems to which the application developer does not have access. The imported 3D data is modeled in third-party tools such as *3DStudio Max*. In this kind of software it is not possible to use custom rendering techniques such as those based on points. Regarding the interaction, these frameworks are capable of reading VR devices, but the developer needs to develop all the support to enable interaction techniques for a particular device.

Low level frameworks, such as VrJuggler [7], Avango NG [16], and INVRS [6] are based on third-party scene graphs for rendering. If the application to be converted uses one of these scene graphs, then it will probably be more easily converted

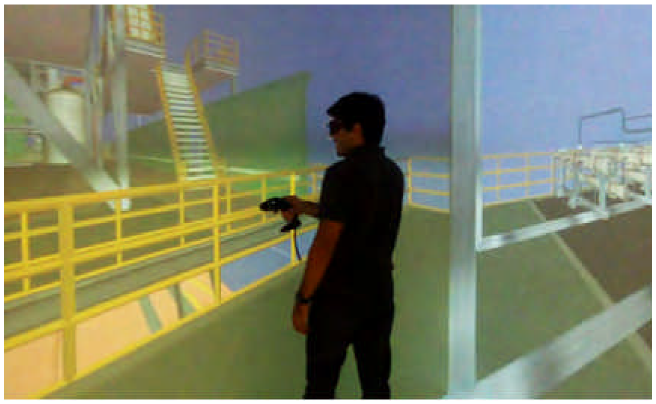


Fig. 1. LVRL usage example: CAVE with Flystick2 support.

using one of these frameworks because they will provide the clustering and multi-view features. However, in general it will be necessary to keep the two versions of the same software, one for the conventional desktop and the other for immersive environments. This happens because these frameworks need to control the main loop of the application to do the synchronized rendering across all screens. Furthermore, related to 3D interaction, they have the same problem as authoring tools, that is, they provide raw data from the devices, and the developer is responsible for handling data in order to use it in the application. This requires the developer to have a deeper knowledge on VR devices.

Cavelib [9] was the first library for immersive environments, and was created together with the first CAVE. It has been maintained and sold up to the present time. Its only difference to the frameworks mentioned above is that it is not tied to a scene graph. Its strategy is to assume that all the dynamic information demanded by the rendering will be in a protected memory area. This memory is shared and can be distributed among computers. However, although it is not tied to a specific scene graph, it still requires that development be guided to it. Additionally, it also needs the camera interaction to be implemented at the same way as the other libraries because it only provides a layer for input devices with an interface similar to VRPN [19].

This coupling with scene graphs or strategies of memory usage of all the solutions mentioned is closely linked with the distribution of rendering across multiple machines. The manager of this distribution needs to know what to distribute and thus be able to synchronize the drawing on different machines. However, despite providing an important feature, it adds a very strong constraint to the framework. At the same time, there are other solutions for generic distributed rendering [11], [14], which do not have the native calculation of multiple viewpoints, but provide the same functionality to applications that are able to internally calculate the multiple points of view.

LVRL already provides the support for camera manipulation and calculations of multiple points of view. In this way we can reduce the efforts in creating applications that are used both on desktop and immersive environments. Instead of delivering just input devices events, the framework provides the required matrices to configure the application camera.

3. LVRL

The framework provides tools for programmers without knowledge of virtual reality algorithms to convert a desktop program into a multi-modal program with 3D interface that runs on desktop and immersive environments just by changing the configuration of the environment at run-time. For this we created libraries that perform the following tasks: (1) assist the existing rendering system to create the virtual cameras for multiple views, (2) provide camera manipulator designed to work in both types of environments in the same way with some devices and provide an interface to *Wand*,, which is a generalization of the mouse to 3D interaction. Below is presented the description of the framework's overall architecture and each of its libraries.

3.1 ARCHITECTURE

The framework consists of six libraries, illustrated in Figure 2. A dependency diagram can be seen in Figure 3. Two of these are mathematical libraries without any dependencies that also can be used alone. They are *VrFrustum* and *VrManipulators*. The other four are related to

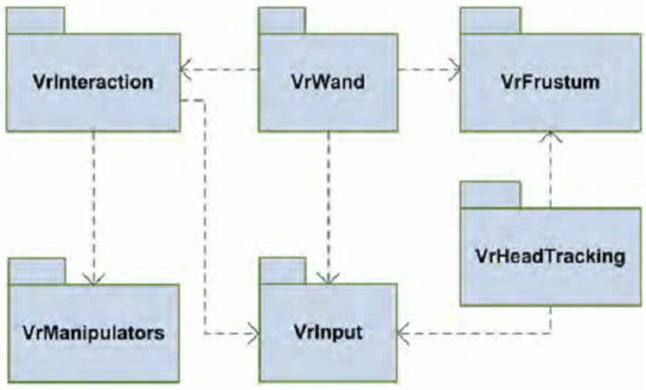


Fig. 3. The diagram shows the dependencies between the libraries.

input data; they read and process input events in order to properly interact with the camera. They can be used with other frameworks, such as *VrJuggler*, but it is necessary to use the input library, *VrInput*, rather than the one provided by other frameworks.

VrFrustum is responsible for evaluating the parameters of cameras from different viewpoints corresponding to each of the screens of an immersive system. It uses the physical configuration of screens and the user's head position. In desktop mode, it generates a single camera for the display.

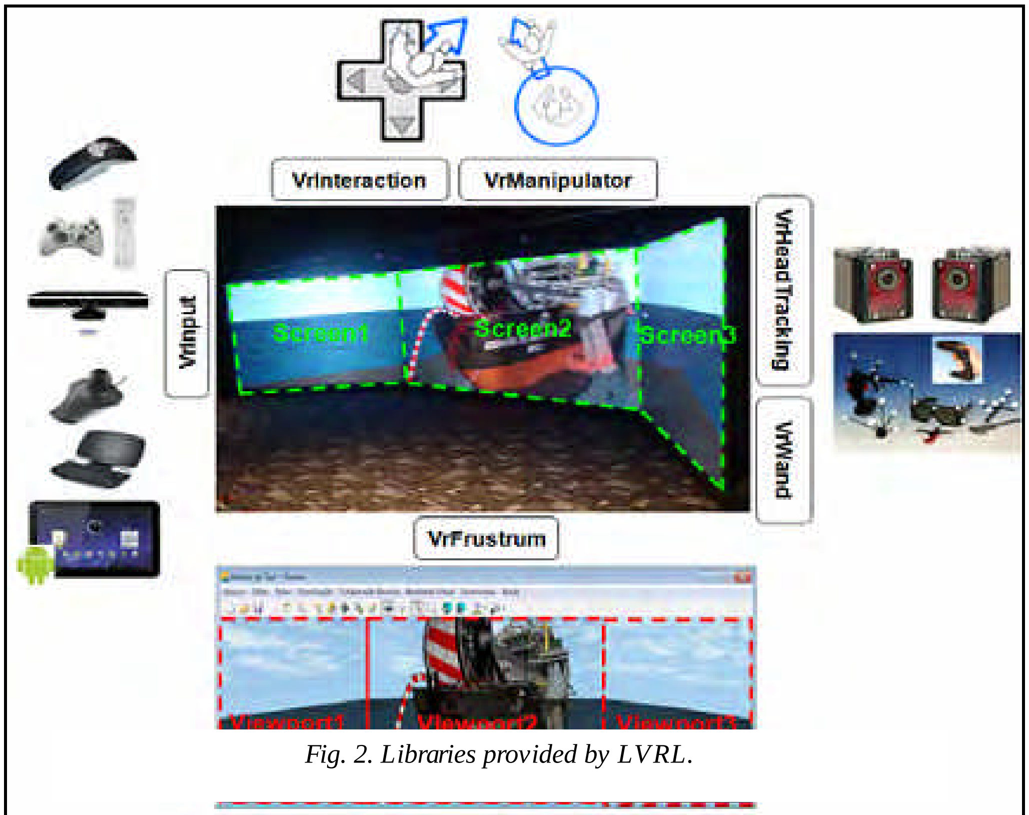


Fig. 2. Libraries provided by LVRL.

VrManipulators contains a set of camera manipulators that are designed to work with input from the mouse and keyboard in the same way as with input from VR devices.

The other four libraries work to capture and interpret input devices: *VrInput*, *VrInteractio*, *VrHeadTracking*, and *VrWand*.

VrInput is responsible for managing input devices.

VrHeadTracking monitors the device that evaluates head position and feeds *VrFrustum* with this position.

VrInteraction is responsible for interpreting input events and transforming them into camera manipulation. This library provides a view matrix that positions and orients the camera in the scene. When *Vrinput* generates an event of one of the input devices supported by *VrInteraction*, it interprets this event according to the previously specified mapping between device and a manipulator Then it calls *VrManipulator* to make the mathematical calculations that update the current view matrix.

Finally, *VrWand* informs the direction at which the user is pointing the device. In VR mode, devices generally provide the absolute position within the immersive environment as well as the orientation of the device. Using *VrInteraction* to transform the position of the input device's coordinates within the immersive environment (real world) to virtual world coordinates, the library is able to evaluate the direction vector and the position of the ray.

Details of each of these libraries are explained below.

3.2 VRINPUT

The *VrInput* library is responsible for accessing input devices, like mice, keyboards, joysticks, and trackers. Currently, the following devices are supported:

- Mouse and keyboard;
- The wiimote control;
- Iphone and Ipad;
- Android devices;
- The flystick2 control ;
- BraTrack tracker;
- SpaceBall;
- Joysticks (only on Windows);
- Kinect (only on Windows);

In addition to the devices listed, there is also support for the VRPN library [19].

VrInput is based on events that are created when a new input is generated by the device. Through a manager, *VrInput* sends these events to the application, which then decides the actions to be performed.

Integration with the application is done through a single interface, responsible for creating the access to devices. To ensure stability and correct functionality, the life cycle of the devices is internally controlled by the *VrInput*, and therefore direct access to devices is prohibited.

To prevent concurrent access to multiple devices, only one instance of *VrInput* is allowed per application. At the time *VrInput* is created, it executes a secondary process which will directly access the devices. Communication between this process and the *VrInput* is done through inter-process calls using shared memory. This architecture ensures that in case of errors on the device access, the application will continue running, since only the secondary process will be affected.

Once the devices to be used are defined, an operation is performed only when the function *ProcessEvents* is called. At this point *VrInput* communicates with the secondary process and verifies if there are pending events. Also, it certifies that the secondary process is still running. If not, it tries to recover the previous state by running the secondary process again and creating access to the defined devices. If this recovery operation is possible, the application will continue to run unaware. Only if the recovery fails are the errors reported to the application.

VrInput also supports loading and saving all of its configuration. This file can be created for a specific environment and can be shared with all applications that use LVRL. It is usually created by the maintainer of the system, and the developer just needs to load this configuration file to activate all devices of a specific system.

3.3 VRFRUSTUM

The *VrFrustum* is responsible for configuring multiple cameras required for immersive environments based on the position and orientation of a main camera. This feature is common in VR frameworks. However, the main difference proposed by *LVRL* is that *VrFrustum* does not need to have access to the camera abstraction in the application. For performing calculations it simply needs to know the geometry of the projection system and the position of the user's head in each frame.

In case of a desktop environment with a single monitor the calculation is restricted to a single screen. However, in the case of a CAVE-like environments, most commonly composed of more than 3 screens, "N" points of view will have to be calculated corresponding to the "N" system screens. In the case of a stereoscopic system it would be "2 x N".

The configuration of the projection system is done through an XML file that adds the parameters inherent in the configuration of a specific immersive environment. This file is common to all applications that work in this environment and is created by the environment maintainer who has the knowledge about the system. Thus the developer does not have to know any information about the visualization system where the application will run. As a main information, this file contains the 3D position of 4 corners of the screens. A screen can be a monitor, a projector, or two projectors (for projection with passive stereo support). This information, along with user head position provided by the tracker system, is enough for the *frustum* calculation to multiple cameras.

The configuration file also contains information about the mapping between the different video outputs of projectors and their regions (or screens) of the projection system (i.e. mapping viewports). The viewport to the right eye is optional, and can be used when the system uses active stereo projection. If stereoscopy is not supported,

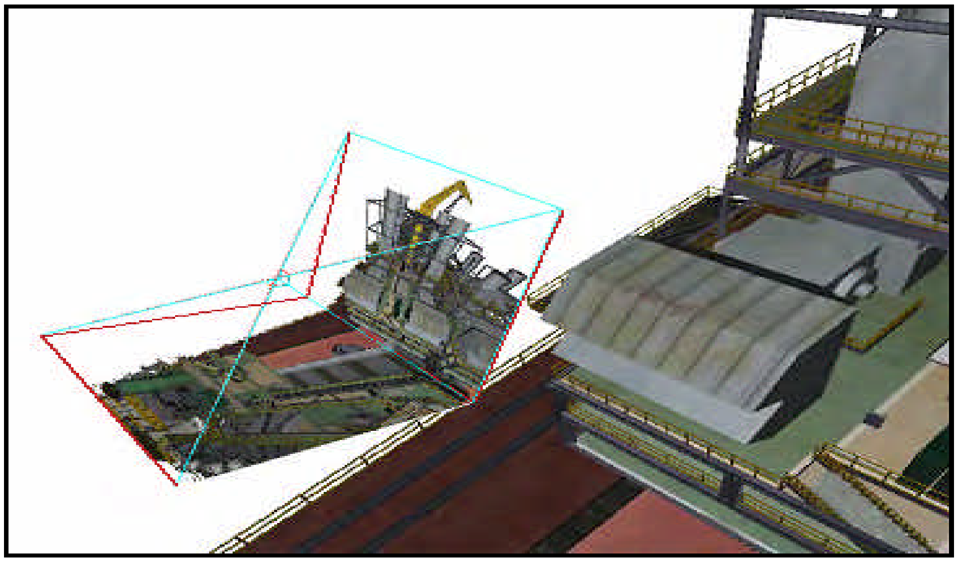


Fig. 4. Visualization of a immersive system defined

only one viewport is necessary. In the case where the immersive system needs to work in a distributed way, in addition to viewport a unique ID is necessary to allow mapping a machine to a specific viewport.

The last property that a screen can have is the reference screen. This property exists because clipping planes *near* and *far* can only be set for one screen. The other screens need to be calculated based on the reference screen. This allows clipping planes to fit together without discontinuities.

Other important information for creating a 3D scene in immersive environments consists of the Pivot and the *OffsetTracking*. *Pivot* is a 3D position in the scene and has two functions. The first is to be used as the standard position of the user's head in case the system does not support head tracking. The second is the origin of the coordinate system of the view matrix. It is provided by the library being used for positioning the cameras in immersive environments.

The *OffsetTracking* is a 3D position that represents the origin of the coordinates system of the tracking equipment. We consider that the coordinates of all systems have the same orientation, and the description of the corners are in the same units provided by the equipment tracking. The meter is a measure commonly used.

For the application, the result of *VrFrustum* is composed only by the projection and view matrices concerning the different views defined in configuration file. The application is responsible for properly applying these matrices to their cameras abstraction. Thus the *VrFrustum* is not dependent on any type of visualization library. This facilitates its use in legacy environments, where the whole display engine is already defined. Figure 4 shows the final arrangement of frustums calculated by *VrFrustum* based on a configuration file for an L-format projection system.

3.4 VRHEADTRACKING

VrHeadTracking is the simplest library in *LVRL*. Its sole function is to pass to *Vrfrustum* the position and orientation from the sensor that is tracking the user's head. *Vrfrustum* then recalculates the frustums based on the new position of the head, thereby creating the effect of head tracking. To use *VrHeadTracking* the application needs only to communicate what tracking device is being used.

3.5 VRMANIPULATORS

The *VrManipulators* library contains the implementation of the manipulators. A manipulator creates and modifies a view matrix which if applied to the application's camera, produces a specific interaction behavior. At the time this paper is written, four manipulators were implemented:

- *Fly*: simulates the flight behavior. Among the available manipulators, it provides the least restrictive navigation. The user can point the camera in any direction to observe the scene. At the same time, translations can be done by referencing the direction the camera is pointing. It is also possible to change the velocity while the navigation is being performed.
- *Examine*: this manipulator is used for object inspection. The camera is rotated around a point, called the pivot. This is usually set at the center of the object, but can also be any point in the scene. How the pivot is chosen is a specific application operation. A zoom tool is also provided by this manipulator.
- *Walk*: this manipulator has a behavior similar to the fly, with the restriction that translation movements are only performed in such a way that the camera remains on the floor plane. This way, the camera acts like it is walking through the scene. For the cases where different geometries are not connected (for example, two floors that do not have stairs connecting then), a

"jump" tool is provided. When the user starts a jump, the camera moves vertically. When the jump is stopped, the camera falls back to the floor plane. The location of the floor plane is determined by the application. Usually, this is done by continuous choosing in the down direction.

- *Rail*: In this manipulator the translation movements can only be made on a specific path, previously chosen by the user. The camera can move in both directions of this path and, at the same time, can be pointed in any direction. This manipulator was projected for presentations where specific characteristics of a scene have to be showed. At any location of the path, another manipulator can be activated to perform other operations during the presentation. When the *rail* manipulator is set back, the camera is smoothly taken to its last position in the path so the presentation can continue.

The manipulators have configuration parameters that can be adjusted by the application. Currently, the following parameters are available:

- *Navigation velocity*: the velocity the camera will move.
- *Rotation velocity*: the velocity the camera will perform rotations.
- *Up vector*: the up vector of the scene. In our implementation, all rotations are constrained in such a way that the camera's up vector is always aligned with the scene up vector. This proved to be an important feature as it helps users to better perceive camera orientation, specially in immersive visualization systems.
- *Pivot point*: the point used by the *examine* manipulator to perform rotations.
- *Floor point*: a point located in the floor plane. In conjunction with the up vector, it is used to find the floor plane equation that will be used by the *Walk* manipulator.
- *Path*: a set of points that establish the path used by the *Rail* manipulator.
- *Unlock/Lock rotation about the right vector*: We noticed that sometimes it is appropriate to limit the freedom of camera rotation in order to facilitate the completion of some tasks. Furthermore, in some types of visualization environments, certain rotations can cause confusion to the user. This is the of rotations around the CAVE's right vector. When this occurs, the user has the impression that the scene is "bent" on some of the screens. In order prevent these situations, this parameter allows the application to lock the rotations about the right axis.
- *Unlock/Lock rotation about the up vector*: it has the same effect of the previous parameter, except that it operates in the rotations around the up vector.

The implemented manipulators provide the basic support for navigation and inspection in 3D visualization applications and were designed to operate in environments ranging from the standard desktop to immersive environments like CAVEs. Currently, we are working on the creation of new manipulators, some of these being just extensions of the existing ones.

3.6 VRINTERACTION

VrInteraction is a manager for interaction operations. It controls how input devices influence the manipulators as well as the process of switching between them. It is a combination of the libraries *VrInput*, *VrManipulators*, and a set of mappings in a way to provide a minimal interface that applications can use to support basic 3D interaction techniques without much effort. A mapping is a relation between an input device and a manipulator. For example, it can specify that a mouse drag will cause a rotation in the *examine* manipulator. A mapping listens to input events via the *VrInput* library and maps theses events to specific actions on the manipulators provided by the *VrManipulators* library.

Currently there are pre-defined mappings for mouse and keyboard, and the *Wiimote* and *Flystick2* controls (Figure 1). They allow basic navigation and inspection on 3D visualization systems. Specifically, the mouse and keyboard devices are used in conjunction in order to define the behavior for standard desktop interaction. These pre-defined mappings were created for all the manipulators of *VrManipulators* and, so there are currently 12 mappings: 4 for desktop (mouse and keyboard), 4 for the *Wiimote* control and 4 for the *Flystick2* control. New mappings using Kinect, Android, iPhone, and iPads devices are being developed. The resources used by these pre-defined mappings should not be accessed directly by applications, as this may cause conflicts in which the resources are used for two different tasks simultaneously. For this reason, each mapping was developed to take advantage of the device features as best as possible using a minimal resources of it. For example, there are 6 buttons on the *Flystick2*, but no more than 2 are used. The remaining buttons are left to the developer, who can use them via the *VrInput* library to perform specific tasks in the application.

The main advantage of the pre-defined mappings is to minimize the work of the developer who wants to create a VR applications. S/he doesn't need to worry about code related to VR interaction because it is already embedded in *VrInteraction*. It is also not necessary to think about specific treatment for VR environments since the manipulators and mappings are designed to work on both desktop and immersive environments. This allows *VrInteraction* to have a minimal interface, where the application needs only to inform the device and manipulator to be used in a given moment. This contributes to reduce the cost of creating new 3D applications and converting legacy applications into immersive ones. Finally, the pre-defined mappings provides a standard interaction for applications that use it. To the user, this simplifies the process of learning, since s/he will not to have to learn a new way of interaction when starting to use a different application. The use of mappings, however, is also flexible: if developers do not want to use one of these pre-defined mappings, they can disable it and a register a new one created by them.

When the application requests a change of manipulator, *VrInteraction* ensures that this operation does not result in a discontinuity in the interaction. This is done through the use of *transitions*. A *transition* is an interpolation between two different poses of camera which aims to ensure a smooth interaction. For example, if the *rail* manipulator is chosen, *VrInteraction* creates a transition to take the camera's current position to the starting position of the path defined on *rail* manipulator. The presence of these transitions is important to prevent situations where the user can get disoriented.

Although depending on the *VrInput* library, an application can have multiple instances of *VrInteraction*. Each of these provides a view matrix which the application applies to his camera abstractions. This view matrix is the result of the combination of the inputs with the manipulators. This way *VrInteraction* can be used in collaborative VR applications, where each user is able to control a different camera [10].

3.7 VRWAND

Actions such as selecting objects or options on menus are made by mouse when in desktop environment. Immersive environments, however, usually do not have mice, and thus different devices should be used. Such devices, called *wands*, are generally adapted controls with sensors whose position and orientation can be monitored by a tracking system.

Through the orientation and position of these devices, applications can obtain a ray which can be used as a virtual 3D pointer. The *VrWand* function is used to calculate the position and orientation of this ray within of the virtual world defined by the application. Via *VrInput*, *VrWand* obtains the position and orientation of the wand in the tracker system coordinates. With the view matrix provided by *VrInteraction*, these data are transformed to world coordinates and passed to the application, which can perform actions like selection, as well as the rendering of the ray.

4. Examples of Use

In this section we are going to present six applications that use the presented libraries to adapt to immersive environments.

Algorithm 1 Pos3D render without LVRL calls

```
  Pos3D.getScreenData(
  prjMatrix, mvMatrix, viewportVector )

  setViewport( viewportVector )
  setProjectionMatrix( prjMatrix )
  setModelviewMatrixPos3D( mvMatrix )

  scene.render()
```

Algorithm 2 Pos3D render with LVRL calls

```
  VrInput.processEvents()
  VrInteraction.update()
  VrWand.update()
  VrHeadtracking.update()
  VrInteraction.getViewMatrix( interactionMatrix )

  for i = 1 to VrFrustrum:getNumberOfScreens() do

      VrFrustrum.getScreenData(
      i, prjMatrix, mvMatrix, viewportVector )

      setViewport( viewportVector )
      setProjectionMatrix( prjMatrix )
      setModelviewMatrixPos3D(
      mvMatrix * interactionMatrix )

      scene.render()

end for
```

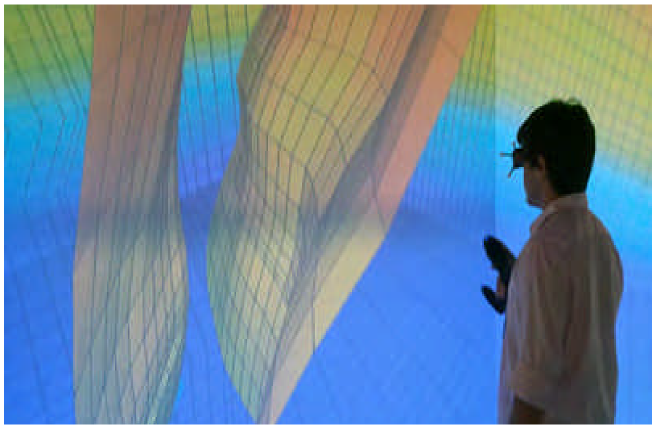


Fig. 5. Pos3D in use on a CAVE

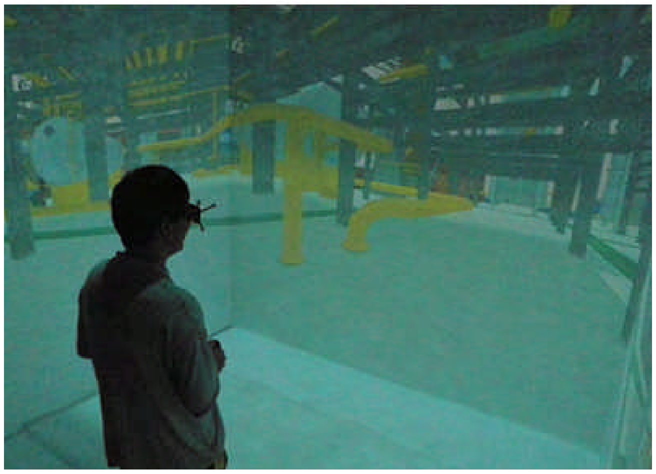


Fig. 6. SimUEP in use on a CAVE.

4.2 SIMUEP

SimUEP is a solution that uses simulators for industrial training. The user controls an avatar that performs tasks on an oil platform. This software was developed using *Unity3D* as game engine and some plug-ins developed in C++. In order to add capabilities that VR software require, it was necessary to use only *VrInput*, *VrFrustum* and *VrHeadTracking* libraries. Because Unity3D is a game engine, it already implement avatar manipulation with joysticks. Thus, the use of *VrInteraction* was not necessary. Figure 6 shows the software in use.

4.3 SIVIEP

SiVIEP is a project under development during the last six years. *SiVIEP* supports a comprehensive visualization of several types of models comprising an oil exploration and production enterprise. For example, it is possible to load from oil platforms to wells and reservoirs in a single scene (Figure 7). This software is based on the *OpenSG* [3] which is a scene graph with distribution features. *OpenSG* support native VR rendering in multiples nodes and multiples screens. In this case it was used by just the *VrInput* and the *VrInteraction* in order to provide the navigation and the support of VR device events, such as *ART Flystick*.

4.4 LVRL Viewer

The LVRL Viewer was created to test all the features of LVRL. It was build upon *OpenSceneGraph* [2] with Qt [4]. The navigation user tests was performed on it. A Screen shot of the LVRL Viewer is shown in Figure 8.

4.5 Cay Viewer

Cay Viewer is a component-based viewer for offshore data that is under development. It will support all kinds of offshore visualization, including simulations. It is written with a mix of C++ and Lua scripts. It also uses a new component framework, *Coral Lib*. Then LVRL was used with a binding for Coral. As a result, LVRL can also be used in Lua scripts. Cay uses *OpenSceneGraph* [2] and a embedded solution to multi-node render based in *RPC* and *ZeroMQ*. Cay uses all LVRL.

4.6 Environ

Environ [17] is a visualization software for massive engineering models. This software was originally developed for desktop, and a few years ago there was a need to add VR features on it. Our group has developed several versions of third-party VR libraries for it that converged into LVRL.

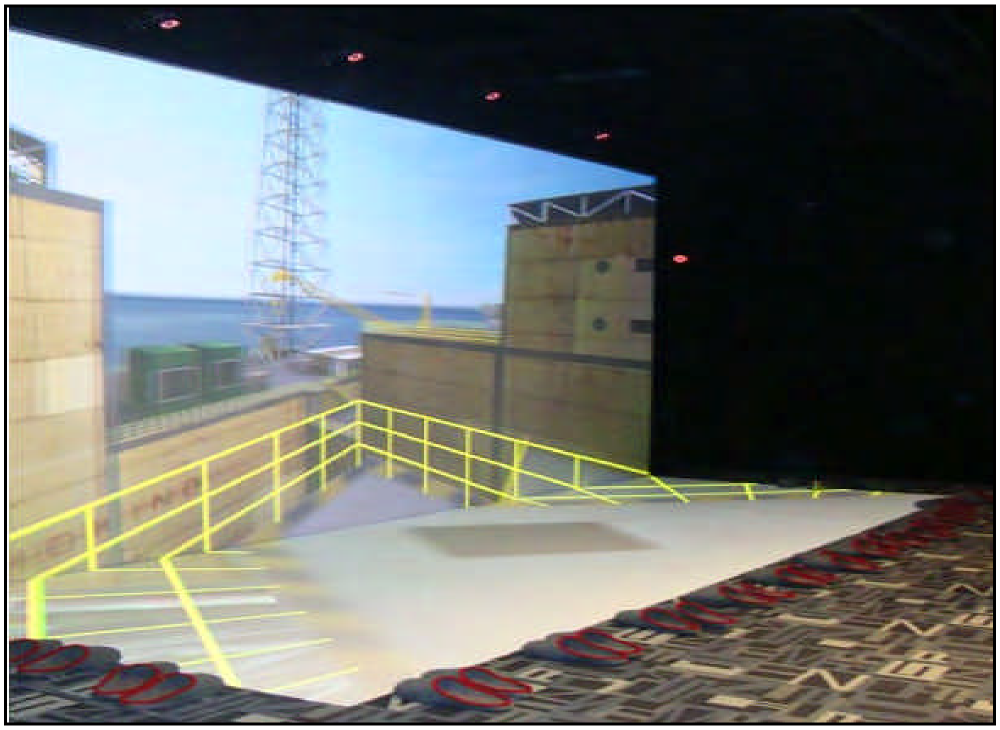


Fig. 7. SiViEP in use on an L-Shape environment.

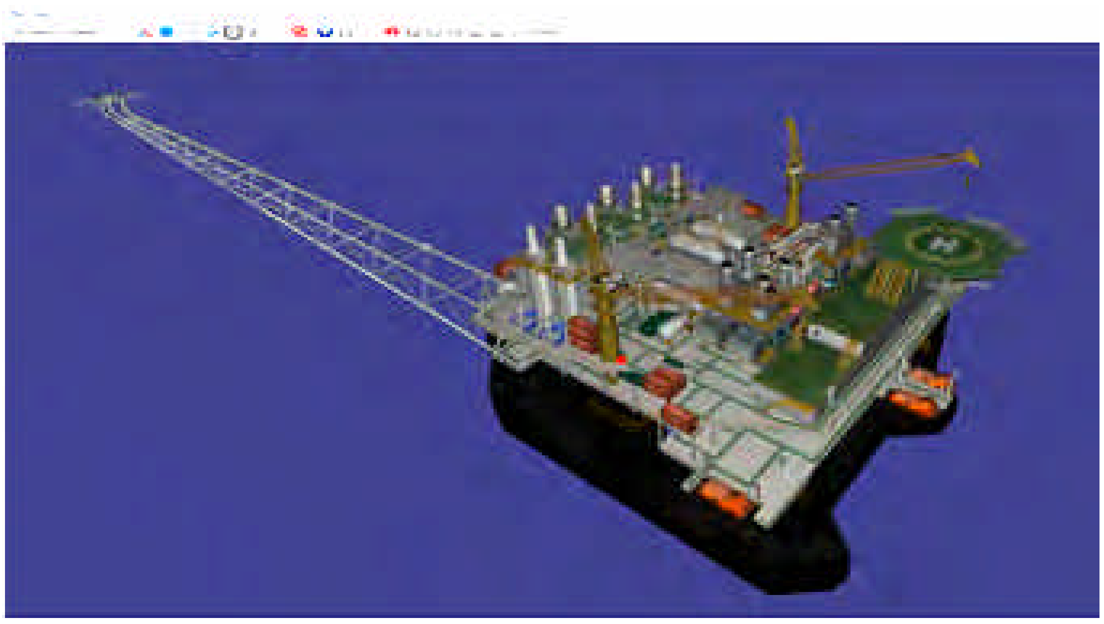


Fig. 8. Screen shot of oil platform in the LVRL Viewer.

5. Results and Discussion

In this section we discuss the main features and contributions of the LVRL framework.

5.1 Architecture Analysis

1) Transparent programming interface: Using the entire framework, the application has contact with the libraries *VrInteraction*, *VrWand*, *VrInput*, *VrHeadTracking* and *VrFrustum*. In the previous sections it was shown that they all have the same results in both desktop mode and VR. Thus developers can program their applications without worrying about details relating to the type of visualization environment. The best examples of this feature was the use of LVRL in the Pos3D by a non-VR programmer.

2) Change desktop to immersive mode at runtime: With the design of *VrInteraction* and *VrFrustum* it is possible to change from desktop to immersive format and vice versa during execution by just making a method call. To reconfigure the screens, it is enough to just load a new configuration file in *VrFrustum*. To use a VR device instead of the traditional mouse and keyboard, you need only tell *VrInteraction* which device to use. This feature proved to be very important in the Cay Viewer, LVRL Viewer, and Environ because a single software version needs to be maintained, saving hours of development..

3) Non-intrusive: The execution control still belongs to the application and not to LVRL, which can be consulted whenever the application needs it. Developers can use whichever graphics engine they want, and it is not necessary for the application to be rewritten to work with LVRL. In *Pos3D* and *SiViEP*, for which the development started before the existence of LVRL, this feature was shown. Just a few parts of software were replaced and the idle control of the application did not change the ownership.

4) Multiplatform: All the libraries have no system dependences and were developed to run on multiple operating systems. Currently Windows and Linux platforms are supported. The exception lies only in *VrInput*, which depends on the drivers for various devices. Thus some devices are not supported on all platforms. Despite this dependence, the design based on a central manager and several device readers whose engagement are events based only on text messaging, allows *VrInput* to be used on multiple platforms, even when a device has no driver for the platform.

5) Hardware independence: With the use of *VrInput* the presence or absence of a device driver does not influence the use of the library. Only the reader related to that device is unavailable.

6) Portable: The implementation of the framework uses only native types of C++. In this way, we can easily port it to C, Unity3D, and Lua, thereby showing that the architecture of the framework does not impose any restriction on portability between languages. The SimUEP and the Cay Viewer show this feature.

7) Compatible with distributed rendering: Although the library does not have distributed rendering primitives, *VrFrustum* configuration provides the necessary tools to achieve this goal. The Cay Viewer shows this feature using an embedded distribution solution. The master node runs the *VrInput* instance and the *VrFrustum* instance. Then it sends to each slave node that camera pose ready to be rendered.

5.2 Users Tests

The LVRL already provides implementations for some camera manipulators through the *VrManipulators* and *VrInteraction* modules. The *fly* examine and *walk* manipulators are commonly used by several 3D applications, and most of the users have experience with at least one of these tools.

To prove that the manipulators in LVRL are suitable to perform common navigation and interaction tasks, we compared our solution with a commercial one. For this, we used Autodesk's NavisWorks as a commercial solution and the LVRL Viewer as described in the previous section. The main reasons for choosing NavisWorks are its broad use in engineering design review, the contributions of Autodesk in the 3D interaction research area, and the positive feedback provided by some of our users about NavisWorks.

Our tests consisted in asking the user to perform specific navigation tasks using both applications. As an example, we asked the user to "go to a place A", or even "find object B, examine it, and count how many objects of type C that object B has". The tasks were composed in a way that the user had to use the *fly* as well as examine and walk manipulators in combination to complete the tasks. A group of 8 people participated, identified as P1 to P8. These people frequently use 3D visualization software and were familiar with 3D interaction tools.

Before starting the test, both options were presented to the user and time was given to him/her to become familiar with both applications. Then the user received the tasks to be executed, and the time was measured for each of the two applications. To reduce the impact of the learning effect, the order in which the applications were used was different for different users: if the user P1 starts the test with the LVRL Viewer, then the next user P2 would starting the test using the NavisWorks.

The results are shown in Table I. It shows for each user the times spent on the execution of the tasks for the LVRL Viewer and NavisWorks respectively. The last column shows the relationship between the time using NavisWorks relative to the time using LVRL Viewer. As can be seen, the time required to perform the tasks was lower using the manipulators provided by LVRL for all users and, in some cases, it was almost half of the time spent when using *NavisWorks.*

As main factors for these results, we identified some differences between the LVRL manipulators and NavisWorks ones. The *fly* provided by NavisWorks, for example, does not maintain the camera aligned with the world up vector. Thus, as the user changes the camera orientation, the view tends to be tumbled in NavisWorks, which caused discomfort for all users. Furthermore, to change the navigation speed the users had to use a menu interface. In desktop mode the LVRL allows change in the navigation speed using the mouse scroll, so the user does not have to stop the navigation to do this.

TABLE 1

Comparison of LVRL with commercial Solution

User	LVRL time	Navis Works time	Navis Works/ LVRL
P1	116	212	1.83
P2	182	232	1.27
P3	136	218	1.60
P4	223	251	1.12
P5	145	254	1.75
P6	122	165	1.35
P7	172	333	1.93
P8	173	253	1.46

Furthermore, the constraint of maintaining the camera aligned with the world up vector is applied to all manipulators. Another benefit pointed out by users is the fact that the manipulators of LVRL are similar with known camera control techniques offered in 3D games. This was pointed out not only by younger users. Even older users said they already had some contact with 3D games, either by themselves or by way of their children. Thus the initial learning phase to understand the functioning of the manipulators of LVRL became faster as it was facilitated by previous experience that most users already had.

In addition to the above tests, other tests were also performed during development of LVRL to determine how to use the available input devices to control the manipulators. These tests were made with the participation of a group of users with mixed profiles, including advanced users (with prior experience in using 3D applications) and beginner users (with little or no prior experience in using 3D applications). These tests consisted of presenting, for a specific input device, different ways to control the manipulators. At the end each user was asked which was better. Users were also encouraged to give their opinion during the test execution, and may also indicate different ways to control the camera from those presented. From these results, the standard mapping in the module *VrInteraction* was determined.

6. Conclusions and Future Work

6. Conclusions and Future Work

LVRL demonstrated to be a framework with the necessary characteristics to be used in most scientific visualization applications from our group.

The non-intrusive architecture and transparent interface programming are the main features that allow non-VR programmers to convert or develop new applications for immersive environments. We believe LVRL helps to decrease the time spent on this process. Thus, developers can focus on other aspects instead of spending time on implementing infrastructural aspects. According to Greenberg [13], this time saving encourages more people to evaluate their application on VR environments. As a result, more new and innovative immersive applications may appear. Consequently the VR area may grow faster.

The main future work to be done is a study of ergonomics and usability to achieve the best form of interaction for the mappings present in *VrInteraction*. In addition, there are ongoing studies on new forms of interaction involving devices like kinect, smartphones and pads.

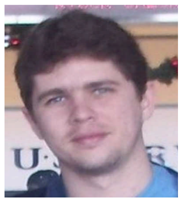
ACKNOWLEDGEMENT

Tecgraf is a laboratory mainly funded by Petrobras. Alberto Raposo receives a grant from CNPq.

REFERENCES

[1] Barco. http://www.barco.com/.
[2] Openscenegraph. http://www.openscenegraph.org.
[3] Opensg. http://www.opensg.org/.
[4] Qt - cross-platform application and ui framework. http:// www.qt.nokia.com/.
[5] DASSAULT SYSTEMES. 3DVIA Virtools. http://www. virtools.com, April 2012.
[6] Anthes, C., Satomi, M., Wilhelm, A., Sommerer, C., and Volkert, J. Space trash : An interactive networked virtual reality installation. In 11th Virtual Reality International Conference (VRIC 09) (Laval, France, April 2009), pp. 107–118.
[7] Bierbaum, A., Just, C., Hartling, P., Meinert, K., Baker, A., and Cruz-Neira, C. Vr juggler: A virtual platform for virtual reality application development. In Proceedings of the Virtual Reality 2001 Conference (VR'01) (Washington, DC, USA, 2001), IEEE Computer Society.
[8] Carvalho, M. T., Martha, L. F., and Celes, W. Pos3d: A generic post-processor to 3d finite-volume models (um pos-processador generico para modelos 3d de elementos finitos). In Proceedings of the Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI'97) (1997), IEEE Computer Society.
[9] Cruz-Neira, C., Sandin, D. J., Defanti, T. A., Kenyon, R. V.,and hart, J. C. The cave: audio visual experience automatic virtual environment. Commun. ACM 35, 6 (June 1992), 64–72.
[10] Dos Santos, I. H. F., Soares, L. P., Carvalho, F., and Raposo, A. A collaborative virtual reality oil & gas workflow. IJVR 11, 1 (2012), 1–13.
[11] Eilemann, S., Makhinya, M., and Pajarola, R. Equalizer: A scal-able parallel rendering framework. IEEE Transactions on Visualization and Computer Graphics 15, 3 (May 2009), 436–452.
[12] Gascon, J., Bayona, J. M., Espadero, J. M., and Otaduy, M. A. Blendercave: Easy VR authoring for multi-screen displays. SIACG 2011: V IBERO-AMERICAN SYMPOSIUM IN COMPUTER GRAPHICS (2011).
[13] Greenberg, S. Toolkits and interface creativity. Multimedia Tools and Applications 32 (2007), 139–159.
[14] Humphreys, G., and Hanrahan, P. A distributed graphics system for large tiled displays. In Proceedings of the conference on Visualization '99: celebrating ten years (Los Alamitos, CA, USA, 1999), VIS '99, IEEE Computer Society Press, pp. 215–223.
[15] Humphreys, G., Houston, M., N G, R., Frank, R., Ahern, S., Kirchner, P. D., AND Klosowski, J. T. Chromium: a stream-processing framework for interactive rendering on clusters. In Proceedings of the 29th annual conference on Computer graphics and interactive techniques (New York, NY, USA, 2002), SIGGRAPH '02, ACM, pp. 693–702s
[16] Kuck, R., Wind, J., Riege, K., and Bogen, M. Improving the avango vr/ar framework: Lessons learned. Workshop VR/AR 2008 (2008).
[17] Raposo, A., Santos, I., Soares, L., Wagner, G., Corseuil, E., and Gattass, M. Environ: Integrating vr and cad in engineering projects. IEEE Comput. Graph. Appl. 29, 6 (Nov. 2009), 91–95.
[18] Soares, L. P., Raffin, B., and Jorge, J. A. Pc clusters for virtual reality. IJVR 7, 1 (2008), 67–80.
[19] Taylor, II, R. M., Hudson, T. C., Seeger, A., Weber, H., Juliano, J., and Helser, A. T. Vrpn: a device-independent, network-transparent vr peripheral system. In Proceedings of the ACM symposium on Virtual reality software and technology (New York, NY, USA, 2001), VRST '01, ACM, pp. 55–61.
[20] Wang, F. Research on virtual reality based on eon studio. In Proceedings of the 2010 Fourth International Conference on Genetic and Evolutionary Computing (Washington, DC, USA, 2010), IEEE Computer Society, pp. 558–561.

AUTHOR BIOGRAPHIES



Daniel Trindade Researcher at Computer Graphics Technology Group - Tecgraf/PUCRio. MSc in Computer Science at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio). Graduated in Computer Engineering at the Federal University of Espírito Santo, Brazil. Current interests: Virtual Reality, 3D interaction, and computer graphics.

Email: danielrt@tecgraf.puc-rio.br

Lucas Teixeira holds a Master's degree in Computer Graphics and a Computer Engineering degree from Pontifical Catholic University of Rio de Janeiro(2010/2007). He is currently a researcher at the Computer Graphics Group - Tecgraf in this university. He has experience in Computer Vision, Augmented and Virtual Reality.

Email: lucas@tecgraf.puc-rio.br

Manuel Loaiza holds a Phd. in Computer Science from Pontifical Catholic University of Rio de Janeiro, Brazil. He is currently a researcher at the Tecgraf, Computer Graphics Technology Group in PUC-Rio, working in several projects at Petrobras. Current interests areas: Computer Vision, Virtual and Augmented Reality, 3D Reconstruction and Computer Graphics.

Email: manuel@tecgraf.puc-rio.br

Felipe Carvalho holds PhD in Computer Science from Pontifical Catholic University of Rio de Janeiro, Brazil. He is currently a researcher at the Computer Graphics Technology Group (Tecgraf) working in several projects at Petrobras. His research interests include 3D Interaction techniques, Virtual and Augmented Reality, and Development of Nonconventional Devices.

Email: kamel@tecgraf.puc-rio.br

Alberto Barbosa Raposo is Assistant Professor at the Dept. of Informatics / PUC-Rio, project coordinator at Tecgraf/PUC-Rio and FAPERJ researcher. DSc in Electrical/Computer Engineering at the State University of Campinas, Brazil. Current interests: Virtual Reality, 3D interaction, groupware, HCI, and computer graphics, with more than 120 publications in these areas. Projects supported by: Petrobras, CNPq, FINEP, FAPERJ and RNP. Distinguished young scholar, PUC-Rio and NVIDIA Academic Partner.

Email: abraposo@tecgraf.puc-rio.br

Ismael H. F. dos Santos has worked for Petrobras (Brazilian Oil company) since 1987. He holds a PhD in Computer Graphics, specialized in Virtual Reality, from Pontificial Catholic University of Rio de Janeiro and collaborates with the Computer Graphics Technolgy Group - TecGraf since 2000. He also has a master's degree in Applied Mathematics from the Federal University of Rio de Janeiro and he also lectures in many software development courses both internal and external to Petrobras.

Email: ismael@tecgraf.puc-rio.br