interactive networked virtual reality system

—

Going Immersive Tutorial

Christoph Anthes and Roland Landertshamer

September 14, 2009

Abstract

The *inVRs* framework was created to ease the design and the development of Networked Virtual Environments. Typically such applications make use of stereoscopic multi-display installations and tracking systems as well as a variety of exotic input devices. This document provides a brief introduction on how to use *inVRs* with multi-display installations and a variety of tracking systems. It demonstrates how to develop a simple object viewer using basic navigation, interaction on typical Virtual Reality installations with a hands-on example.
After going through the tutorial, the user will be able to navigate through a dataset and change a few parameters of the dataset.

Contents

Chapter 1

Introduction

The *inVRs* framework [AV06] was designed to ease the creation of Networked Virtual Environments (NVEs). Considering the concept of NVEs not only desktop environments, but rather truly interconnected Virtual Reality (VR) applications are falling in this category. This class of applications makes use of stereoscopic displays, which are often implemented as multi-display installations. These displays are normally equipped with 3D tracking systems to allow for intuitive user interaction.
Many libraries for accessing tracking systems and even more types of such systems exist. The input used in a VR application is often not solely provided by position tracking, all types of arbitrary devices can be used to generate input to VR applications.
In order to represent the user in an NVE often 3D models, or so called avatars, are integrated in the scene to display remote users. Depending on the amount of given sensors they can be animated in a fairly realistic manner.
Users of this tutorial should already be familiar with the first part of the *inVRs* tutorial series the *Medieval Town Tutorial* in order to understand the basic concepts of the framework. With the basic knowledge of the *inVRs* framework the readers will now be able to extend their knowledge into the immersive sector.
Many aspects of the initial tutorial will be simplified and abstracted and additional aspects like input devices, displays and coordinate system will be explained in depth.

1.1 Tutorial Overview

Initially the reader will learn how to wrap up the cumbersome configuration setup which was explained in detail in the *Medieval Town Tutorial* in order to get an insight in the frameworks inner workings. The tutorial will provide a brief introduction into the basics of immersive displays as well as 3D tracking systems and common VR input devices. It will be described how to configure the displays and how to write own input device drivers, by using existing drivers and libraries. The different coordinate systems will be used in conjunction with the user representations to display remote users.
The following topics will be explained in the context of the *inVRs* framework:

- Wrapping Functionality
- Immersive Displays
- Using the Input Interface
- Coordinate Systems
- Tracking and Avatars

At the end of the tutorial the readers should be able to develop their own interactive multi-user applications for CAVEs [CNSD+92], Head-Mounted Displays (HMDs) [Sut68], curved displays and similar devices. Position tracking systems can be used for interaction and are incorporated to display remote users. An abstract virtual world will be displayed and acts as a proxy for arbitrary VEs.

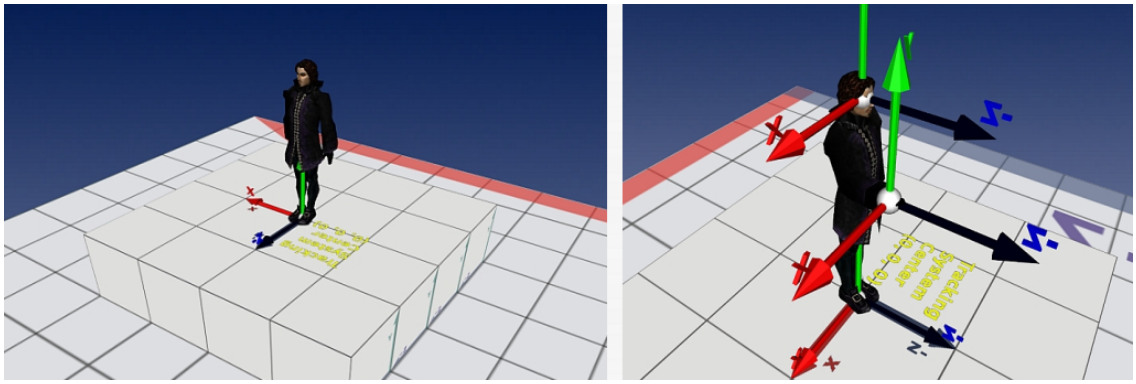

Figure 1.1: The Going Immersive Application

Figure 1.1 illustrates the resulting application form the completed *Going Immersive Tutorial*. The left side of the figure gives an overview on the scene, while the right side of the figure shows the coordinate systems of the avatar.

1.2 Outline

The chapters of this tutorial cover the following topics:

- Chapter 2 - Wrapping Functionality
  The cumbersome setup of an *inVRs* application as seen in the first tutorial is replaced by using a wrapper class. This application base is introduced and explained in this chapter. The generic abstract class application base is used as a super class for the actual implementation for OpenSG scene graphs.

- Chapter 3 - Immersive Displays
  A variety of immersive displays is introduced as is the setup of the framework in order to demonstrate the configuration of these displays. The *inVRs* framework uses the CAVE Scene Manager in connection with OpenSG in order to generate graphics output on arbitrary rectangular display panes. It is explained how to configure your VR display and how to interconnect it to the *inVRs* framework.

- Chapter 4 - Using the Input Interface
  These previously mentioned displays often come with a variety of specific input devices. The reader will learn how to interconnect already available or own devices to *inVRs*, by writing specific drivers for the input interface.

- Chapter 5 - Working with Avatars
  To display remote users avatars can be used and the data gathered from the tracking systems can be mapped on these avatars. Different types of avatars exist. A more advanced avatar will be introduced as the one experienced in the first tutorial.

- Chapter 6 - Coordinate Systems
  Coordinate systems are a key aspect for displaying avatars, and implementing interaction when tracking systems are used. The dependencies of the different world and user coordinate systems will be explained in depth.

- Chapter 7 - Outlook
  The taught aspects of *inVRs* are recaptured and a brief outlook on what else could be explored using the framework is given.

Chapter 2

Wrapping Functionality

In general it is possible to develop an *inVRs* application as described in the first tutorial – the *Medieval Town Tutorial*. Much of the code developed in that tutorial is generic and is already available in a so called `ApplicationBase` which is part of the *inVRs* SystemCore. The application developer can derive from this class to spare the implementation of the generic code parts. Besides the general existing `ApplicationBase` also a scene graph specific implementation the `OpenSGApplicationBase` is available as an additional tool. This class will be used in this tutorial as a basis.

2.1 Using the ApplicationBase

Before we will start with the tutorial let's have a look at the generic `ApplicationBase` class. The key functions which are already provided in the `ApplicationBase` class are described in the following:

- `virtual bool preInit(const CommandLineArgumentWrapper& args)`

  This method is called before the initialization of the application base. It is the first method called in an *inVRs* application after the constructors are executed. The application developer can overwrite this method to insert code which has to be executed before all other parts of the application.

- `virtual void initCoreComponentCallback(CoreComponents comp)`

  This method is called by the `SystemCore` when the components of this class are initialized. It can be overwritten by the application developer to get notifications before each component is initialized.

- `virtual void initInputInterfaceCallback(ModuleInterface* moduleInterface)`

  This method is called by the `InputInterface` when the modules of this class are initialized. It can be overwritten by the application developer to get notifications before each module of the `InputInterface` is initialized.

- `virtual void initOutputInterfaceCallback(ModuleInterface* moduleInterface)`

  This method is called by the `OutputInterface` when the modules of this class are initialized. It can be overwritten by the application developer to get notifications before each module of the `OutputInterface` is initialized.

- `virtual void initModuleCallback(ModuleInterface* module)`

  This method is called by the `SystemCore` when the general modules are initialized. It can be overwritten by the application developer to get notifications before each module is initialized.

- `virtual bool disableAutomaticModuleUpdate()`

  This method can be overwritten by the application developer if the updates of the different modules should be done manually in the application. By default the update of all registered modules is done automatically by the ApplicationBase class. If this class is overwritten and it returns true then the automatic module update is omitted and the virtual method `manualModuleUpdate()` is called instead (see next line).

- `virtual void manualModuleUpdate(float dt)`

  This method is called automatically when the `disableAutomaticModuleUpdate()` method described above was overwritten accordingly. The default implementation of this method in the `ApplicationBase` class does nothing, so ensure to overwrite this method if you want to do the module updates manually.

Other functions have to be implemented by the application developer:

- `virtual std::string getConfigFile(const CommandLineArgumentWrapper& args)`

  The main configuration file has to be passed to *inVRs*. It is typically loaded from a fixed path or passed as a command line argument.

- `virtual bool init(const CommandLineArgumentWrapper& args)`

  The method is called after the initialization of all *inVRs* components, interfaces and modules. It is intended to be used by the application developer for the initialization of the main application.

- `virtual void run()`

  This method is called by *inVRs* after all initialization steps were finished and the runnable components like the `EventManager` and the `Network` module were started. In this method the application developer should implement the main application loop. The implemented main loop has to call the `ApplicationBase::globalUpdate()` method every loop iteration in order to update the *inVRs* components.

- `virtual void display(float dt)`

  This method is called by the `ApplicationBase::globalUpdate()` method in order to update the main application. The application developer should prefer this method for updates compared to the implementation in the main loop because some important *inVRs* updates like the `Controller` or the `TransformationManager` were executed before this method is called.

- `virtual void cleanup()`

  This method should be implemented in order to clean up the application. It is called by the `ApplicationBase::globalCleanup()` method.

Other methods which have to be called by the application developer:

- `bool start(int argc, char** argv)`

  This method starts the application. It should be called out of the main method after an instance of the application object was created.

- `void globalUpdate()`

  The method updates the *inVRs* components and forwards the update-command to the inherited `display` method. It must be called out of the application main loop.

- `void globalCleanup()`

  The method cleans up the *inVRs* components. It must be called by the application before finishing. This method automatically forwards the command to the inherited `cleanup` method with which the user can clean up the main application.

Besides the provided methods several member variables can be used when inheriting from the `ApplicationBase`:

- `SceneGraphInterface* sceneGraphInterface`

  This member variable points to the used SceneGraphInterface. If no SceneGraphInterface is used the pointer value is NULL.

- `ControllerManagerInterface* controllerManager`

  This member variable is a pointer to the ControllerManager. If no ControllerManager is used the pointer value is NULL.

- `NetworkInterface* networkModule`

  This member variable points to the Network module. If no Network module is loaded the pointer valus is NULL.

- `NavigationInterface* navigationModule`

  This member variable points to the Navigation module. If no Navigation module is loaded the pointer points to NULL.

- `InteractionInterface* interactionModule`

  This member variable points to the Interaction module. If no Interaction module is loaded the pointer points to NULL.

- `User* localUser`

  This variable points to the User object for the local user.

- `CameraTransformation* activeCamera`

  This variable is a pointer to the camera transformation object of the local camera.

Using the `ApplicationBase` class allows for developing applications without having to care about the main *inVRs* components. Although this reduces the lines of code which have to be written there is still much to implement, e.g. for the window management, or the display methods for rendering the scene.

2.2 Using the OpenSGApplicationBase

To further simplify the application development the `OpenSGApplicationBase` was developed. This class is inherited from the basic `ApplicationBase` class and implements additional functionality for window management, rendering and input device support. In this helper class the decision whether immersive displays with the help of the CAVE Scene Manager are used for output or a simple GLUT window is used is taken.
The key functions which are already provided in the `OpenSGApplicationBase` or it's derived classes are described in the following:

- `virtual bool preInitialize(const CommandLineArgumentWrapper& args)`

  This method is called before the initialization of the application base. It is the first method called in an *inVRs* application right after OpenSG was initialized (via `osgInit()`). The application developer can overwrite this method to insert code which has to be executed before all other parts of the application. **NOTE**: Take care to not confuse this method with the `ApplicationBase::preInit()` method, since this one is implemented by the `OpenSGApplicationBase` and must NOT be overwritten!

- `virtual void initCoreComponentCallback(CoreComponents comp)`

  This method is called by the `SystemCore` when the components of this class are initialized. It can be overwritten by the application developer to get notifications before each component is initialized.

- `virtual void initInputInterfaceCallback(ModuleInterface* moduleInterface)`

  This method is called by the `InputInterface` when the modules of this class are initialized. It can be overwritten by the application developer to get notifications before each module of the `InputInterface` is initialized.

- `virtual void initOutputInterfaceCallback(ModuleInterface* moduleInterface)`

  This method is called by the `OutputInterface` when the modules of this class are initialized. It can be overwritten by the application developer to get notifications before each module of the `OutputInterface` is initialized.

- `virtual void initModuleCallback(ModuleInterface* module)`

  This method is called by the `SystemCore` when the general modules are initialized. It can be overwritten by the application developer to get notifications before each module is initialized.

- `virtual void cbGlutSetWindowSize(int w, int h)`

  This method is called whenever the size of the window is changed. The application developer can overwrite this functions if this information is needed by the application.

- `virtual void cbGlutMouse(int button, int state, int x, int y)`

  This method is called whenever a mouse button is pressed inside the application window. It can be overwritten to get notifications for mouse button presses. Note that it is recommended to get this information via the *inVRs* `ControllerManager` using the `GlutMouseDevice` instead!

- `virtual void cbGlutMouseMove(int x, int y)`

  This method is called whenever the mouse cursor is moved over the application window. It can be overwritten to get notifications for mouse motion. Note that it is recommended to get this information via the *inVRs* `ControllerManager` using the `GlutMouseDevice` instead!

- `virtual void cbGlutKeyboard(unsigned char k, int x, int y)`

  This method is called whenever a keyboard key is pressed. It can be overwritten to get notifications for keyboard input. Note that it is recommended to get this information via the *inVRs* `ControllerManager` using the `GlutKeyboardDevice` class.

- `virtual void cbGlutKeyboardUp(unsigned char k, int x, int y)`

  This method is called whenever a keyboard key is released. It can be overwritten to get notifications for keyboard input. Note that it is recommended to get this information via the *inVRs* `ControllerManager` using the `GlutKeyboardDevice` class.

Other functions have to be implemented by the application developer:

- `virtual std::string getConfigFile(const CommandLineArgumentWrapper& args)`

  This method must return the url to the main configuration file (usually called `general.xml`). This file is needed by *inVRs* in order to load and configure the core components, interfaces and modules. If the url to the configuration file should be passed via command line then this value can be obtained from the `CommandLineArgumentWrapper` parameter.

- `virtual bool initialize(const CommandLineArgumentWrapper& args)`

  This method can be used in order to initialize the application. The method is called after the *inVRs* components (core, interfaces and modules) were initialized. The parameter of type `CommandLineArgumentWrapper` can be used to read options passed via the command line.

- `virtual void display(float dt)`

  This method is called every application loop cycle and can be used to update the application. The parameter `dt` contains the elapsed time in milliseconds since the previous method call.

- `virtual void cleanup()`

  The method should be used in order to cleanup the application. It is called right before the application is terminated.

Other methods which must be called by the application developer:

- `void setRootNode(NodePtr root)`

  In this method the root node of the scene graph is set in the used `SceneManager`. Depending on the configuration this is either the `SimpleSceneManager` or the `CAVESceneManager`.

Other methods which can be called by the application developer:

- `void setPhysicalToWorldScale(float scale)`

  This method is important when writing applications for VR installations like a CAVE. By calling this method you can set the scale-factor from the physical units provided by your tracking system to the world units used in the application. For example if your tracking systems provides centimeter values and your application is modeled in meters then you should pass 0.01 to this method. Don't forget to call this method when using tracking systems in order to provide a correct visualization.

- `void setNearClippingPlane(float nearPlane)`

  This method allows to set the near clipping plane of your application.

- `void setFarClippingPlane(float farPlane)`

  The method allows to set the far clipping plane of your application.

- `void setStatistics(bool onOff)`

  This method activates or deactivates the display of the `SceneManager` specific statistics.

- `void setWireframe(bool onOff)`

  This method allows to switch between normal and wireframe rendering.

- `void setHeadlight(bool onOff)`

  By calling this method the headlight (default light used in OpenSG) can be activated or deactivated.

- `bool setBackgroundImage(std::string imageUrl, int windowIndex = -1)`

  This method sets the background image for the windows with the passed index. When no window index is passed the image is used as background for all active windows. **NOTE:** Using an image background in OpenSG may reduce the overall performance of your application. Avoid using this method or only use it with low resolution images!

- `void setEyeSeparation(float eyeSeparation)`

  This method allows to set the eye separation when using the `CAVESceneManager` for output.

- `float getEyeSeparation()`

  The method returns the current eye separation.

Besides the already provided methods several member variables which are inherited from the class `ApplicationBase` can be used:

- `SceneGraphInterface* sceneGraphInterface`

  This member variable points to the used SceneGraphInterface. If no SceneGraphInterface is used the pointer value is NULL.

- `ControllerManagerInterface* controllerManager`

  This member variable is a pointer to the ControllerManager. If no ControllerManager is used the pointer value is NULL.

- `NetworkInterface* networkModule`

  This member variable points to the Network module. If no Network module is loaded the pointer valus is NULL.

- `NavigationInterface* navigationModule`

  This member variable points to the Navigation module. If no Navigation module is loaded the pointer points to NULL.

- `InteractionInterface* interactionModule`

  This member variable points to the Interaction module. If no Interaction module is loaded the pointer points to NULL.

- `User* localUser`

  This variable points to the User object for the local user.

- `User* localUser`

  This variable points to the User object for the local user.

- `CameraTransformation* activeCamera`

  This variable is a pointer to the camera transformation object of the local camera.

In order to write your own *inVRs* application using OpenSG your application should inherit from the `OpenSGApplicationBase`.

2.3 Initial Tutorial Application

The first thing which has to be done is to implement a class for the application which inherits from the `OpenSGApplicationBase`. This class is called `GoingImmersive` in our case. The following listing shows the declaration of this class:

```
#include <OpenSGApplicationBase/OpenSGApplicationBase.h>
#include <inVRs/SystemCore/WorldDatabase/WorldDatabase.h>

OSG_USING_NAMESPACE

class GoingImmersive: public OpenSGApplicationBase {
...
}; // GoingImmersive
```

Listing 2.1: GoingImmersive.cpp - Top Part of application

In this tutorial the whole application will be developed in a single **.cpp** file without using a header file. This is done in order to simplify the documentation but could be splitt up into separate a header and source file as well.
After the class is declared we have to define the members of our class. In the first step of our application development the only member variable we need is the url to the main *inVRs* configuration file. This member variable will be initialized in the constructor of the `GoingImmersive` class:

```
class GoingImmersive: public OpenSGApplicationBase {

private:
  std::string defaultConfigFile;    // config file

public:
  GoingImmersive() {
    defaultConfigFile = "config/general.xml";
  } // constructor

...

}; // GoingImmersive
```

Listing 2.2: GoingImmersive.cpp - Top Part of class

Besides the constructor the application class must also contain a destructor. In this destructor the `OpenSGApplicationBase::globalCleanup()` method must be called in order to free all memory reserved by the different *inVRs* components.

```
...
  ~GoingImmersive() {
    globalCleanup();
  } // destructor
...
```

Listing 2.3: GoingImmersive.cpp - Destructor

Since the `OpenSGApplicationBase` is an abstract class several methods have to be implemented in the derived class. The first method which must be implemented is the `getConfigFile()` method. This method must return the url to the main *inVRs* configuration file. The default configuration file is already stored in a member variable. Besides the default configuration we also want to support the user to pass the url to a different configuration file via the command line. Therefore the `CommandLineArgumentWrapper` class can be used. The following implementation shows how to support the passing of the configuration file url via the command line argument `config=...`:

```
...
  std::string getConfigFile(const CommandLineArgumentWrapper& args) {
    if (args.containOption("config"))
      return args.getOptionValue("config");
    else
      return defaultConfigFile;
  } // getConfigFile
...
```

Listing 2.4: GoingImmersive.cpp - getConfigFile()

The next method which has to be implemented is the `initialize()` method. This method is called automatically after all *inVRs* components were initialized. In this method we will set

the root node of the scene graph and the initial transformation of the user in the virtual world. Therefore the member variables `sceneGraphInterface` and `localUser` which are inherited from the `ApplicationBase` class are used:

```
...
  bool initialize(const CommandLineArgumentWrapper& args) {
    OpenSGSceneGraphInterface* sgIF =
      dynamic_cast<OpenSGSceneGraphInterface*>(sceneGraphInterface);
    // must exist because it is created by the OutputInterface
    if (!sgIF) {
      printd(ERROR, "GoingImmersive::initialize(): Unable to obtain
          SceneGraphInterface!\n");
      return false;
    } // if

    // obtain the scene node from the SceneGraphInterface
    NodePtr scene = sgIF->getNodePtr();

    // set root node to the responsible SceneManager (managed by
        OpenSGApplicationBase)
    setRootNode(scene);

    // set our transformation to the start transformation
    TransformationData startTrans =
      WorldDatabase::getEnvironmentWithId(1)->getStartTransformation(0);
    localUser->setNavigatedTransformation(startTrans);

    return true;
  } // initialize
...
```

Listing 2.5: GoingImmersive.cpp - initialize()

Further methods which have to be implemented are the `display`() and the `cleanup`() method. In our current application we don't have to update any information and also don't have to clean up anything, so both methods are empty:

```
...
  void display(float dt) {

  } // display

  void cleanup() {

  } // cleanup
...
```

Listing 2.6: GoingImmersive.cpp - display() and cleanup()

This is all we have to implement in our first `GoingImmersive` class. Finally we need a main method which creates an instance of our application class and starts the application:

```
...
int main(int argc, char** argv) {
  GoingImmersive* app = new GoingImmersive();

  if (!app->start(argc, argv)) {
    printd(ERROR, "Error occured during startup!\n");
    delete app;
    return -1;
  } // if

  delete app;
```

```
  return 0;
} // main
```

Listing 2.7: GoingImmersive.cpp - main method

That was the whole code of the initial application. Additionally to this code a predefined set of configuration files is contained in the tutorial package. A basic introduction to the configuration files was already given in the previous tutorial – the *Medieval Town Tutorial*. Additionally a separate manual on *Configuring the inVRs Framework* can be found on the *inVRs* homepage which describes the single configuration files in detail. Thus the configuration files will not be explained at this point.
When executing this application you will see a predefined scene containing an plane, a coordinate system and the *inVRs* logo. Due to the configured navigation module you can navigate through this environment via mouse and keyboard input.

2.4 Summary

This chapter has given an insight into concepts of wrapping setup of *inVRs* applications. As a generic approach the application base was introduced. Additionally the OpenSGApplicationBase was described as an implementation of the generic approach. By using application bases you should be able to develop basic *inVRs* applications with a few lines of code.
The following chapter will show how to interconnect immersive displays to your *inVRs* application.

Chapter 3

Immersive Displays

The *inVRs* framework is designed to support a large variety of immersive displays. Typically immersive displays share the feature of generating stereoscopic output on multiple display panes. By using OpenSG[1] [Rei02] on the scene graph side the display on such stereoscopic multi-display installations can be considered as an out-of-the-box feature. The implementation of the *inVRs* output interface uses OpenSG for display purposes. The specific multi-display functionality is abstracted and handled by the external tool – the CAVE Scene Manager.
Generating 3D audio is also supported by *inVRs* but is not part of this tutorial. Aspects like haptic displays or motion platforms are so far not covered at all by the framework, although it would be possible to write drivers for such displays and integrate them into output interface.

3.1 Different Types of Immersive Displays

A huge variety of immersive displays exist. *inVRs* is focusing on stereoscopic multi-display installations. Some examples for such displays would be a CAVE or an HMD.
To reduce the problem of multi-display installations there are basically two big setup possibilities. Either the displays are arranged in a curved or dome-like fashion or in some kind of rectangular shape. Although an interesting aspect, this part of the tutorial does not concentrate on the different display technologies, but simply on the different display setups, focusing on how to arrange and configure the display planes.
Many displays are common for VR and some are rather rare. Figure 3.1 illustrates the most prominent VR displays, a CAVE on the left side and an HMD on the right side.

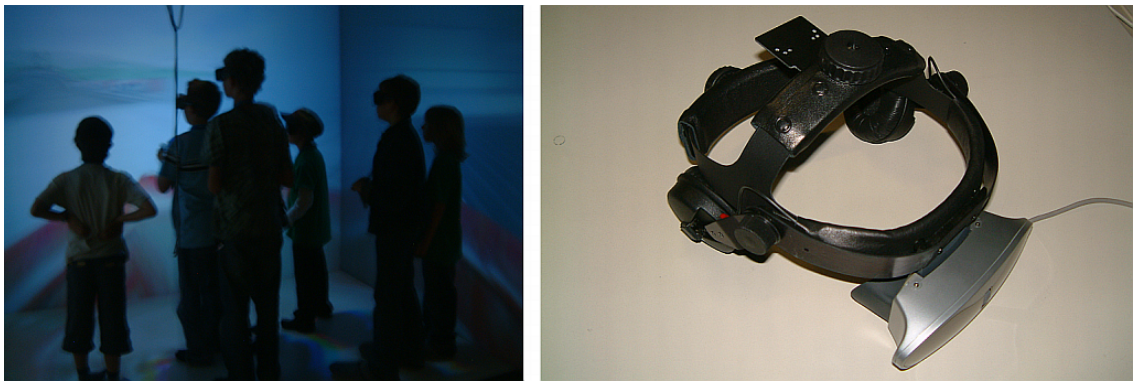

Figure 3.1: A CAVE and an HMD

[1] http://www.opensg.org

Other installations like Curved Screens or Powerwalls are supported as well by the framework. While a powerwall could be used with the most simple setup, curved displays require a bit for math for setting them up properly. Figure 3.2 show two wide spread displays which are commonly used for larger audiences, where the previously introduced displays are more applicable for single users.

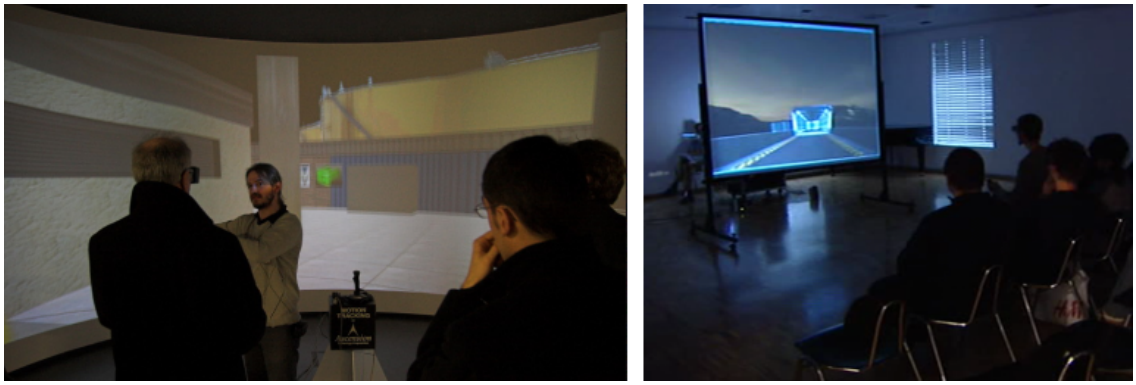

Figure 3.2: A Curved Screen and a Powerwall

There are many other setups like for example the ImmersaDesk [CPS+97] or the Responsive Workbench [KF94].
We will now first have a brief look on how to configure the setup of such immersive displays. In the next step the interconnection of these displays with the *inVRs* framework will be explained in detail.

3.2 Using the CAVE Scene Manager

One of the main tools used by *inVRs* for handling multi-display functionality is the CAVE Scene Manager. The stereoscopic multi-display functionality in general is covered by using OpenSG as a scene graph. The CAVE Scene Manager is simply used for wrapping OpenSG multi-display support. Additionally it offers the parsing of human readable configuration files. It is designed as a counterpart to OpenSG's `SimpleSceneManager`.
It was originally developed by Adrian Haffegee as a side product of his MSc Thesis [Haf04, HJAA05]. It acts as a wrapper around the OpenSG multi-display and clustering functionality. The CAVE Scene Manager is not part of the basic *inVRs* distribution but it can be downloaded and installed as an additional tool. More detail is provided in the *CAVE Scene Manager Manual*. The tool consists of four main header and source files:

- `OSGCAVESceneManager.h`

  This source file contains the main functionality and user API of the scene manager. It allows to attach a scene to main node and interact with it in a similar way than the SimpleSceneManager.

- `OSGCAVEConfig.h`

  This file contain functionality for parsing, loading, and setting the scene managers configuration. The configuration of the scene manager is mainly concerned with display setup.

- `OSGCAVEWall.h`

  This source file deals with the setup of wall displays. Typically several projection panes are used for displaying an immersive scene.

- `appctrl.h`

  This source file contains functionality for starting and shutting down display servers. It wraps as well the setup of the OpenSG `MultiDisplayWindow`

3.3 Configuring the CAVE Scene Manager

The configuration of the CAVE Scene Manager is highly intuitive and can be used to support pretty much every multi-display installation, which consists of rectangular drawing panes.
The configuration takes typically keywords described in the following and a set of keyword-specific parameters separated by spaces. Comments are indicated by a # following the comment.
The most important keywords for the configuration of the CAVE Scene Manager are given in the following list:

- Walls
  This keyword is used to define a list of display panes which are to be used.
- WallDisplay
  A detailed wall configuration is provided after a WallDisplay keyword. Name, display (e.g. :0.1) and resolution for the display pane is provided. Additional offsets can be can be defined. These offsets can become interesting if overlaps of displays are used.
- ProjectionData
  This keyword defines the alignment of the different projection panes.
- DisplayMode
  The display mode can be set either to `mono` for monoscopic display or to `stereo` for stereoscopic display.
- InterocularDistance
  The eye separation is provided as a number and an optional unit following afterwards.
- Origin
  This keyword describes where the coordinate origin in physical space is located. The data is used for correct rendering of the VE.
- CAVEWidth
  In case a CAVE-like display is used the width is provided by this parameter.
- CAVEHeight
  If a CAVE is used the height can be defined by using this keyword.
- Units
  The units can be either meters, centimeters or foot. Typically *inVRs* make use of centimeters as units.

Many more keywords are available for the configuration of the CAVE Scene Manager and respectively tha multi-display setup. They will be explained in depth in the *CAVE Scene Manager Manual*. The source code of the class `OSGCAVEConfig` provides additional details on the configuration of the CAVE Scene Manager.
The following example will give an idea how to setup your immersive display for *inVRs*. It provides the configuration of a typical CAVE setup.

```
###############################################################################
# Specify here which CAVE walls you want to run and in which graphics pipe    #
# walldisplays for the jku-cave (has 4 walls)                                 #
###############################################################################
Walls front left right floor
```

```
##############################################################################
# Display information for walls (pipe # & (optional) window geometry)        #
# window geometry: XDIMxYDIM+XOFFSET+YOFFSET                                 #
# 2006-11-08 ZaJ: new layout                                                 #
##############################################################################
WallDisplay front :1.0  1136x1136+0+0
WallDisplay right :1.1  1136x1136+0+0
WallDisplay left  :1.2  1136x1136+0+0
WallDisplay floor :1.3  1136x1136+0+0

##############################################################################
# 2008-11-24 LeB: new layout                                                 #
#   info on refpoint for JKU CAVE: center of floor plane (X, x=0,y=0,z=0)    #
#                                                                            #
#        P2  +--------------------------+                                   #
#            | x=-155                   |                                   #
#            | y=0                      |                                   #
#            | z=-155                   |                                   #
#            |                          |                                   #
#            |                          |                                   #
#            |            X             |                                   #
#            |                          |                                   #
#            |                          |                                   #
#            | x=-155            x=155  |                                   #
#            | y=0               y=0    |                                   #
#            | z=-155            z=155  |                                   #
#        P1  +--------------------------+ P3                                #
#                                                                            #
# ProjectionData screenx * wall  P1         P2            P3                #
##############################################################################
ProjectionData floor * wall -120 0 120  -120 0 -120   120 0 120 centimeters

##############################################################################
# Display mode - mono or stereo                                              #
##############################################################################
DisplayMode stereo

##############################################################################
#InterocularDistance <distance> <units>                                      #
##############################################################################
InterocularDistance 6.0 cm

##############################################################################
# Origin of coordinates of the CAVE (given in distance to the walls)         #
# distance to left wall      distance to floor      distance to front wall   #
##############################################################################
Origin          120.0                0.0                     120.0  centimeters

# Cave width (& depth)
CAVEWidth       240.0  centimeters

# Cave height
CAVEHeight      240.0 centimeters

##############################################################################
# Cave units for GL coordinates (Meters or feet)                             #
# - units tracking data will be given in                                     #
##############################################################################
Units   centimeters

##############################################################################
# Size of screen & viewing distance - defines simulator viewing frustum      #
##############################################################################
SimulatorView 10 7.5 2

##############################################################################
```

```
# Which type of wand is being used (mouse or PC)                              #
###############################################################################
Wand daemon

###############################################################################
# Type of tracking (birds, polhemus, logitech, mouse, or simulator)           #
###############################################################################
TrackerType daemon

###############################################################################
# Various Settings                                                            #
###############################################################################
HideCursor y
TrackerDaemonKey  4129
ControllerDaemonKey  4128
```

Listing 3.1: GoingImmersive.cpp

3.4 Displaying Virtual Environments

After a configuration file for the CAVE Scene Manager was created we can start to update the initial *GoingImmersive* application to run on the configured display(s). If you haven't created a configuration file yet or you want to test this application on a monoscopic desktop system you can use the configuration file `mono.csm` which is contained in this tutorial.
Before we can start updating the application we have to prepare the CAVE Scene Manager to be able to display the application. Therefore we have to understand how the visualization is realized by this class. The CAVE Scene Manager is separated into two individual parts, the client part and the render server part. The client part is the CAVE Scene Manager class itself. It is used by the application to manage the OpenSG scene graph. In general the CAVE Scene Manager can be used independently with OpenSG offering an extensive user API. We will skip the description of the API since *inVRs* will take care of most of the functionality.
For displaying the scene the render servers are used. The render servers are stand alone applications which are receiving and rendering the scene graph information from the client part of the CAVE Scene Manager via a network connection. They are very similar to the basic OpenSG render servers as introduced in Oliver Aberts' OpenSG Tutorial [Abe04].
In order to run an application on a multi-display system multiple render servers have to be started. One server is used for one display pane. Depending on your system this can either be done by the application automatically or you have to start the servers in advance by hand. The automatic startup of the render server(s) works in general when these can be run on the same host as the main application. This is true for example on a single-display setup which is run on a single host, but also on a multi-display setup when the graphics pipes are directly accessible from the host the application is started (e.g. shared memory systems with multiple X-servers). On cluster systems the render servers usually must run on the single graphics nodes which means that they can not be started automatically by *inVRs*. In this section we will present the configuration for a single monoscopic display with automatic render-server startup but also describe the steps which have to be executed in order to start the servers manually (for use on multi-display sytems based on a cluster).
To be able to start the render server automatically the binary of the render server has to be placed into the folder from where the application is executed. For the monoscopic server the binary is called server-mono, for stereoscopic visualizations you have to use the server-stereo binary. Copy these binaries from `bin` subfolder of your *inVRs* installations into your application directory `GoingImmersive` now. Furthermore these binaries need to find the CAVE Scene Manager library (`libCAVESceneManager.so` on Linux systems, `libCAVESceneManager.dylib` on Mac OSX systems, or `CAVESceneManager.dll` on windows systems). In order to find this library you can either add the *lib* path of your *inVRs* installation directory to your library path environment

variable or you can simply copy the file into the `GoingImmersive` directory.
Now that the binary for the render server is available we have to tell the application to use the CAVE Scene Manager instead of the `SimpleSceneManager`. When writing an application from scratch without using the `ApplicationBase` helpers this must be changed in the source code. Since we are using the `OpenSGApplicationBase` class we can do this by simply adding some entries in the general *inVRs* configuration file `general.xml`.
The `useCSM` entry tells the `OpenSGApplicationBase` to use the CAVESceneManager instead of the default `SimpleSceneManager`. The second argument `csmConfigFile` defines which configuration file should be used. In our case we are using the file `mono.csm` which is included in this tutorial. In the next option the automatic startup of the render servers is configured. If this entry is missing or set to false you will have to start the render server(s) manually. The fourth entry defines the relation between the world coordinates and the units used in the real world. In this application the objects in the virtual world are modeled approximately in the size of meters and the units used in the configuration file `mono.csm` are centimeters, so the scale value must be 0.01. Finally we set a background image for the control window, which is the window where the input goes to. Note that using a background image for the control window can drop performance depending on the size of this image. So if you get render performance problems try to either use lower resolution images or don't use any image at all.

```
<OpenSGApplicationBase>
  <option key="useCSM" value="true"/>
  <option key="csmConfigFile" value="mono.csm"/>
  <option key="startRenderServers" value="true"/>
  <option key="physicalToWorldScale" value="0.01"/>
  <option key="controlWindowImage" value="inVRs_controlwindow.png"/>
</OpenSGApplicationBase>
```

Listing 3.2: XmlSnippets1.xml - Snippet1-1 → general.xml

Additionally we have to add the paths to the location of the CAVE Scene Manager configuration, which is stored in the configuration file indicated in the previous snippet with the attribute `csmConfigFile`:

```
<path name="CAVESceneManagerConfiguration"
    directory="config/outputinterface/cavescenemanager"/>
```

Listing 3.3: XmlSnippets1.xml - Snippet1-2 → general.xml

The path for loading images has to be set as well. We will need it later on for setting a background image.

```
<path name="Images" directory="images/"/>
```

Listing 3.4: XmlSnippets1.xml - Snippet1-3 → general.xml

That's all we have to do in order to run the application using the CAVE Scene Manager. When you start the application now you will notice some additional console output of the CAVE Scene Manager telling you if the render server(s) could be started successfully or the reason why it could not be started. If the render server(s) could not be started automatically you should be able to see the problems on the debug output and try to start the render server(s) manually. You don't even have to restart the application for this because it is waiting until all needed render servers are available before the application continues with the startup.
When the application has started up successfully you can see two windows (assuming you are using the `mono.csm` configuration file), one window is used for the render server and another window which we call the control window.

The control window is the same window which would be used in a `SimpleSceneManager` application to render the scene. Since we are using the CAVE Scene Manager now the rendering is executed on separate windows which are opened by the render servers. The control window is still visible and is used for example to read the GLUT input. If you wan't to control you application and you are using the GLUT input the control window should be the active window.
In order to get rid of the ugly black background we now add a background image to be used in our window. This is done in the following source code snippet by calling the `setBackgroundImage()` method of the `OpenSGApplicationBase` class. Note that the usage of background images in OpenSG can decrease the performance significantly, so don't use images with a high resolution in order to avoid low framerates.

```
    setBackgroundImage("background_128.png");
```

Listing 3.5: CodeSnippets1.cpp - Snippet-1-1 → GoingImmersive.cpp

When building and starting the application now you should see the same result as shown in Figure 3.3:

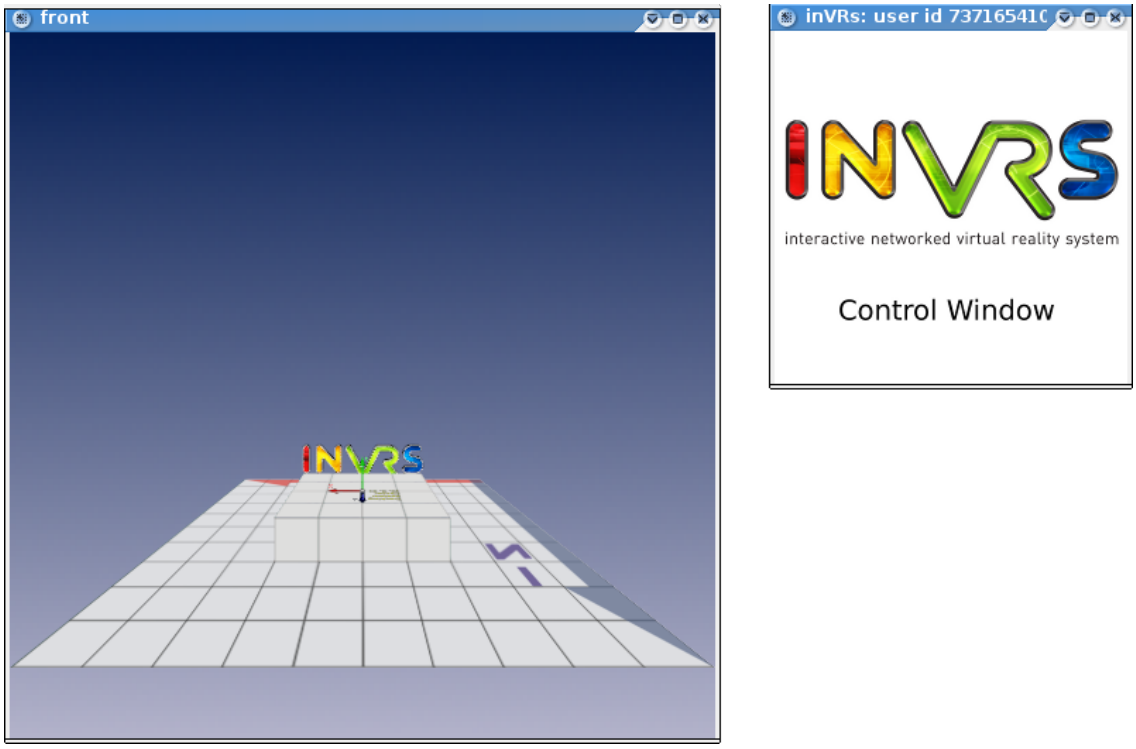

Figure 3.3: Render server window (front) and control window (inVRs)

The left side of the illustration shows the actual application running on a display server while the right side displays a control window which is the input and output window of our application. With the help of the application base and very few code changes and simple configurations we are able to use immersive displays.
Adapting the configuration of your setup only has to be done once. Generic files like the one for monoscopic display or stereo display on desktop system are already provided with the distribution of the CAVE Scene Manager.

3.5 Summary

This chapter has given a very brief overview on common multi-display systems used in the field of VR. CAVEs, HMDs, curved displays and powerwall installations were shortly introduced. As a tool for setting up such displays for the *inVRs* framework the CAVE Scene Manager has been described. An introduction on the configuration of the scene manager was given as well as a description on how to interconnect it to *inVRs*.
You should now be able to display a very simple scene on an arbitrary visual output device. If you have access to a VR installation it might be worth trying to configure the CAVE Scene Manager for it and run your application on the VR system.
In the next chapter we will learn how to use typical VR input devices with our current application. If you have CAVEs or other setups available you should then be able to fully run an *inVRs* application on your VR system.

Chapter 4

Using the Input Interface

A huge variety of input devices can be thought of and a great set of them is available on the market of VR installations. Often devices like wands are used in conjunction with tracking systems in order to allow for user interaction. Considering the tracking as well as the input devices the *inVRs* framework is designed to be totally technology agnostic.
In general the *inVRs* input interface could be extended to support all types of input, like speech or gestures. So far an interface for the abstraction of the traditional input devices is provided which reduces the whole data generated by arbitrary devices to a very simple types of data – axes, buttons and sensors. This abstraction is exposed and later on accessed by the components and the modules of the framework or an application developed with the *inVRs* framework.

4.1 Different Types of Input Devices

Two big categories of devices are available which generate input for VR applications, the input devices providing input intentionally triggered by the user and the tracking systems offering position and orientation data on sensors attached to the user or an input device.
In general a vast amount of input devices and tracking systems exist, based on very different technologies, which are typically accessed by two different kinds of libraries. Either low-level drivers that are used to access the device directly or high-level libraries like VRPN[1] [THS+01], OpenTracker[2] [RS01, RS05], etc. which wrap together many different of these low-level libraries are used to gather the input from the devices.

Figure 4.1: Some Typical VR Input devices

[1] http://www.cs.unc.edu/Research/vrpn/
[2] http://studierstube.icg.tu-graz.ac.at/opentracker/

Figure 4.2 shows some devices which are wide spread in the field of Virtual Reality. On the left side a space mouse, emulating a 6 DOF sensor is shown, the middle illustrates a wand with buttons and joystick and the right side of the figure shows a pair of pinch gloves which generate boolean values on contact of the finger tips.

4.2 Mapping Input on the Abstract Controller

In the *inVRs* framework the actual data of the devices can be either taken directly from the low-level drivers or alternatively from the high-level libraries [3]. To support an input device an interface between the driver or high-level library and the input interface of the framework is either provided by *inVRs* already, as for example for trackD, GLUT, VRPN and an arbitrary UDP controller or it has to be created by the application developer.
The *inVRs* framework chooses a very simplistic approach by taking three different types of data into account as shown in the following list:

- Buttons
  They provide boolean values
- Axes
  They provide values along a slider
- Sensors
  They provide 6DOF position values

These three different types of data can be accessed from the developed application or parts of the framework. Typically models from the `Navigation` module or transition functions which form an interaction technique of the `Interaction` module make use of such exposed abstracted data. User defined modules or the application itself can access the data as well.
For access of the data an abstract `Controller` is configured inside the `InputInterface`. This abstract controller exposes the values to the application parts and the framework. But before the data can be provided a mapping between the devices or better their libraries and the `Controller` has to take place.

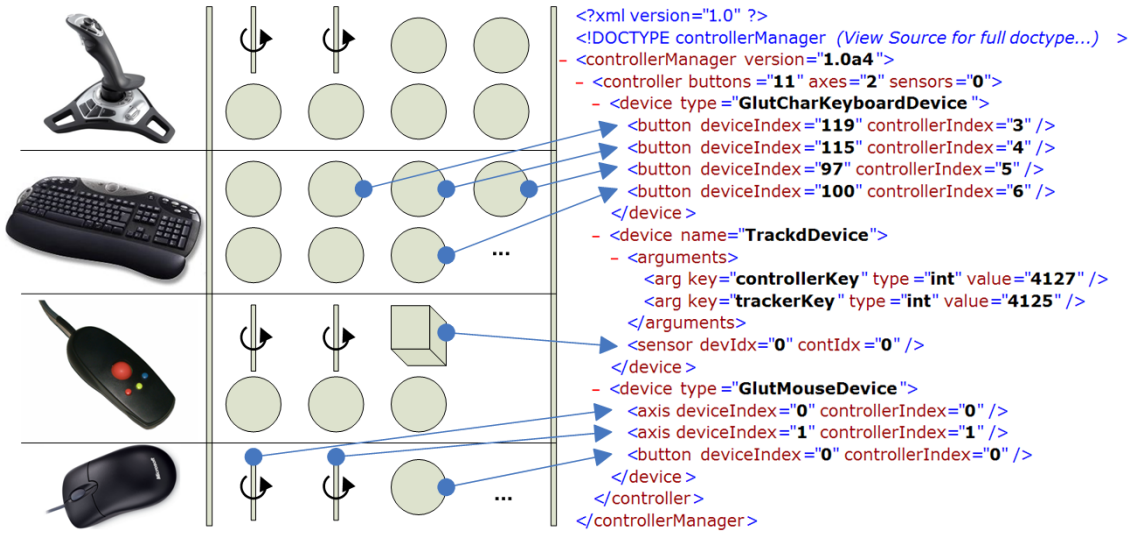

Figure 4.2: An Example Mapping of the Input Interface

[3] if you know exactly what device you intend to use it might make sense to connect it by writing an interface connecting directly to the low-level driver in order to increase performance

An example mapping is given in Figure 4.2. The devices are shown in the left column, the middle column illustrates the possible output they could generate in an abstracted way. The buttons are shown as circles, sensors as boxes and axes are shown as cylinders. The right column of the illustration shows an example XML configuration file which describes how the values provided by the controller are abstracted by a mapping on the abstract controller.
The mapping shows the controller specification with it's <data>-node defining which data can be accessed externally.
Next three groups of devices are identified with their different attributes. Pairs of `deviceIndex` attributes referring to the physical device drivers, and `controllerIndex` attributes, which indicate the abstract controller data, implement the mapping between the used library, indicated by the name of the wrapper in the `name` attribute of the <device>-node, and the `Controller` of the input interface.
Once such a `controller` is defined it's data can be accessed via the following functions:

- `int getButtonValue(int idx)`

  With this function call the current value of the button can be requested. On the button additional callbacks can be registered which are trigged depending on the button state.

- `float getAxisValue(int idx)`

  Returns the value of the axis with the given index. Positive and negative floating point values are valid.

- `SensorData getSensorValue(int idx)`

  Provides a `SensorData` object containing transformation data about the sensor with the given index.

- `int getNumberOfButtons()`

  Returns the amount of registered buttons on the abstract controller.

- `int getNumberOfAxes()`

  Returns the amount of registered axes on the abstract controller.

- `int getNumberOfSensors())`

  Returns the amount of registered sensors on the abstract controller.

4.3 Writing own Devices for the Abstract Controller

Developing a device for *inVRs* is pretty straightforward. The newly developed device has to inherit from the class `InputDevice` and implement the functions for polling the values for the buttons, axes and sensors. Additionally functions for receiving the available amount of these data types have to be implemented. As an example we are going to develop a device which uses the VRPN library.
But first let's have a look at the base class `InputDevice` which we have to inherit from. The following methods have to be implemented when inheriting from the `InputDevice`:

- `int getNumberOfButtons()`

  This method must return the number of buttons which are provided by this device.

- `int getNumberOfAxes()`

  Must return the amount of available axes provided by the device.

- `int getNumberOfSensors()`

  Must return the amount of available sensors provided by the device.

- `int getButtonValue(int idx)`

  Must return the value of the button with the passed index. If the device does not provide a button with this index it must return 0.

- `float getAxisValue(int idx)`

  Must return the value of the axis with the passed index. If the device does not provide an axis with this index it must return 0.

- `SensorData getSensorValue(int idx)`

  Must return the value of the sensor (translation and orientation) with the passed index. If the device does not provide a sensor with this index it must return the predefined value `IdentitySensorData`.

- `void update()`

  This method is called by the `Controller` class once a frame in order to update the values of the input device. It can be used for example in order to read the input values from the low level library and copy the values into the member variables of the input device.

Besides the pure virtual methods the `InputDevice` class provides some methods which can be called by inherited classes:

- `void acquireControllerLock()`

  Call this method if you want to update button, axis or sensor values inside your input device from another thread or outside of the `update`() method. By default the `update`() method should be used to update the input values of the device. But for example when using a callback-based mechanism to gain the input values you have to lock the `Controller` class to avoid conflicts with reading and writing the new input values. This can be done by calling this method.

- `void releaseControllerLock()`

  This method must be called in order to release the controller lock again after the method `acquireControllerLock`() was called.

- `void sendButtonChangeNotification(int buttonIndex, int newButtonValue)`

  This method must be called whenever the state of a button has changed. It then forwards the notification to the `Controller` class which itself sends notifications to all registered listeners that the button value has changed.

These are all methods you have to cope with when implementing a new input device for *inVRs*. Now let's take a look at a specific implementation, the `VrpnDevice`. This class allows to gather button, analog (axis) and tracker (sensor) data from a VRPN device. It therefore uses the VRPN callback mechanism and stores these values in internal data structures. These values are then provided to the *inVRs* application via the methods inherited from the `InputDevice` class. The source code for this device which is described in the following can be found in the *inVRs* sources in the subfolder `tools/libraries/VrpnDevice`.
At first we will have a look at the header file:

```
#ifndef VRPNDEVICE_H_
#define VRPNDEVICE_H_

#include <vrpn_Button.h>
#include <vrpn_Tracker.h>
#include <vrpn_Analog.h>
#include <set>

#include <inVRs/InputInterface/ControllerManager/InputDevice.h>
```

```
/*****************************************************************************
 * InputDevice class for reading values from the Vrpn library.
 */
class VrpnDevice : public InputDevice {
public:
  /**
   * Constructor
   */
  VrpnDevice(std::string deviceId, unsigned numSensors, unsigned numButtons,
      unsigned numAxes);

  /**
   * Destructor
   */
  virtual ~VrpnDevice();

  /**
   * Returns the number of buttons provided by the input device
   */
  int getNumberOfButtons();

  /**
   * Returns the number of axes provided by the input device
   */
  int getNumberOfAxes();

  /**
   * Returns the number of sensors provided by the input device
   */
  int getNumberOfSensors();

  /**
   * Returns the value of the button with the passed index
   */
  int getButtonValue(int idx);

  /**
   * Returns the value of the axis with the passed index
   */
  float getAxisValue(int idx);

  /**
   * Returns the value of the sensor with the passed index
   */
  SensorData getSensorValue(int idx);

  /**
   * Updates the values of the VrpnDevice
   */
  void update();

  /**
   * Returns if the VrpnDevice was successfully initialized
   */
  bool isInitialized() const;

  /**
   * Callback method for the tracker
   */
  static void VRPN_CALLBACK trackerPosQuatCallback(void *userdata,
      const vrpn_TRACKERCB trackerData);

  /**
   * Callback method for the buttons
   */
  static void VRPN_CALLBACK buttonCallback(void *userdata, const vrpn_BUTTONCB
```

```
      buttonData);

  /**
   * Callback method for the analog input data
   */
  static void VRPN_CALLBACK analogCallback(void *userdata, const vrpn_ANALOGCB
      analogData);

private:

  /**
   * Initializes the device
   */
  void initializeDevice(unsigned numSensors, unsigned numButtons, unsigned numAxes)
      ;

  /**
   * Update tracker data
   */
  void updateTracker(const vrpn_TRACKERCB trackerData);

  /**
   * Update button data
   */
  void updateButton(const vrpn_BUTTONCB buttonData);

  /**
   * Update analog data
   */
  void updateAnalog(const vrpn_ANALOGCB analogData);

  std::vector<int> buttonValues;
  std::vector<float> axisValues;
  std::vector<SensorData> sensorValues;
  std::set<int> buttonCallbackWarnings;
  std::set<int> axisCallbackWarnings;
  std::set<int> sensorCallbackWarnings;

  /// defines if
  bool initialized;
  // ID for the device
  std::string deviceId;
  /// member for reading the tracker data
  vrpn_Tracker_Remote* tracker;
  /// member for reading the button data
  vrpn_Button_Remote* button;
  /// member for reading the axis data
  vrpn_Analog_Remote* analog;
}; // VrpnDevice

/******************************************************************************
 * Factory class for the VrpnDevice
 */
class VrpnDeviceFactory : public InputDeviceFactory {
public:

  /**
   * Destructor
   */
  virtual ~VrpnDeviceFactory() {}

  /**
   * Creates a new VrpnDevice if the passed className matches
   */
  virtual InputDevice* create(std::string className, ArgumentVector* args = NULL);
}; // VrpnDeviceFactory
```

```
#endif /* VRPNDEVICE_H_ */
```

Listing 4.1: VrpnDevice.h

In this file we define two classes, the `VrpnDevice` class which is inherited from the abstract `InputDevice` class and a factory class for this device called `VrpnDeviceFactory`. The `VrpnDevice` class implements all pure virtual functions of the `InputInterface` class. Additionally the class provides the `isInitialized`() method which returns whether the `VrpnDevice` was initialized successfully or not. Furthermore the class contains three static methods which are used for VRPN callbacks in order to update the input values of the device (this methods will be described in detail later in this section).
Besides the public methods the class also contains 4 private methods, one for initializing the device and three other methods in order to update the internal variables.
The `VrpnDevice` class contains the following member variables:

- `std::vector<int> buttonValues`

  In this vector the class stores the values of the buttons provided by this input device.

- `std::vector<float> axisValues`

  In this vector the values of the axes provided by this input device are stored.

- `std::vector<SensorData> sensorValues`

  In this vector the values of the sensors provided by this input device are stored.

- `std::set<int> buttonCallbackWarnings`
  `std::set<int> axisCallbackWarnings`
  `std::set<int> sensorCallbackWarnings`

  These sets are used by the class to avoid printing multiple warnings for individual buttons/axes/sensors which are updated but not provided by the device. This could be the case when the VRPN library sends updates for more buttons, axes or sensors than configured in the `VrpnDevice`. Since these members are only for debug output we can ignore them here.

- `bool initialized`

  This variable indicates whether the initialization of the `InputDevice` was successful or not.

- `std::string deviceId`

  In this variable the VRPN device identifier is stored (e.g. trackingDevice@serverHost).

- `vrpn_Tracker_Remote* tracker`

  This variable is used for accessing the tracking data provided by the VRPN device.

- `vrpn_Button_Remote* button`

  This variable is used for accessing the buttons provided by the VRPN device.

- `vrpn_Analog_Remote* analog`

  This variable is used for accessing the analog values (like axes) provided by the VRPN device.

This is all we have to know about the header file. Let's now have a look at the source file to see how the implementation looks like.
The first method which is called from the `VrpnDevice` is the constructor. The parameters needed for the constructor are the device identifier for the VRPN device and the number of provided buttons, axes and sensors. The constructor then directly calls the `initializeDevice`() method which tries to establish the connection to the VRPN device.

```
VrpnDevice::VrpnDevice(std::string deviceId, unsigned numSensors, unsigned
    numButtons,
    unsigned numAxes) :
  initialized(false),
  deviceId(deviceId),
  tracker(NULL),
  button(NULL),
  analog(NULL) {

  initializeDevice(numSensors, numButtons, numAxes);
} // VrpnDevice
```

Listing 4.2: VrpnDevice.cpp - Constructor

In the `initializeDevice()` method the first thing which happens is to create objects for connecting to the tracker, button and analog data provided by the VRPN device with the identifier stored in the `deviceId` variable. After the objects are created the vectors for storing the button, axis and sensor values are initialized.
In the next step the static callback methods for the VRPN objects are registered by calling the `register_change_handler()` methods. This allows the VRPN library to notify the `VrpnDevice` whenever a value has changed. The first parameter which is passed to this method is the pointer to the current class instance. This parameter is later on used in the static callback method to identify the `VrpnDevice` object which has registered the callback (in case that multiple `VrpnDevice`s are used). The second parameter identifies the static callback method which will be called. Finally the `initializeDevice()` method checks if any of the VRPN objects could be created and sets the `initialized` variable accordingly. This variable can then be read by calling the `isInitialized()` method (which is done by the `VrpnDeviceFactory` later).

```
void VrpnDevice::initializeDevice(unsigned numSensors, unsigned numButtons,
    unsigned numAxes) {
  tracker = new vrpn_Tracker_Remote(deviceId.c_str());
  button = new vrpn_Button_Remote(deviceId.c_str());
  analog = new vrpn_Analog_Remote(deviceId.c_str());

  sensorValues.resize(numSensors);
  for (int i=0; i < (int)numSensors; i++) {
    sensorValues[i] = IdentitySensorData;
  } // for

  buttonValues.resize(numButtons);
  for (int i=0; i < (int)numButtons; i++) {
    buttonValues[i] = 0;
  } // for

  axisValues.resize(numAxes);
  for (int i=0; i < (int)numAxes; i++) {
    axisValues[i] = 0;
  } // for

  if (tracker) {
    tracker->register_change_handler(this, &VrpnDevice::trackerPosQuatCallback);
  } else {
    printd(WARNING, "VrpnDevice::initializeDevice(): unable to open vrpn_Tracker!\n
        ");
  } // else

  if (button) {
    button->register_change_handler(this, &VrpnDevice::buttonCallback);
  } else {
    printd(WARNING, "VrpnDevice::initializeDevice(): unable to open vrpn_Button!\n"
        );
  } // else
```

```
  if (analog) {
    analog->register_change_handler(this, &VrpnDevice::analogCallback);
  } else {
    printd(WARNING, "VrpnDevice::initializeDevice(): unable to open vrpn_Analog!\n"
        );
  } // else

  if (!analog && !button && !tracker)
    initialized = false;
  else
    initialized = true;
} // initializeDevice

...

bool VrpnDevice::isInitialized() const {
  return initialized;
} // isInitialized
```

Listing 4.3: VrpnDevice.cpp - initializeDevice()

Now that the device is initialized let's have a look at the static callback methods. Each callback method has a similar implementation.
At first the passed `userdata` argument is casted into a `VrpnDevice` pointer. This pointer is the same which was passed as first parameter at callback registration time in the `initializeDevice()` method.
Afterwards the update method for the appropriate data type of the obtained device object is called.

```
void VRPN_CALLBACK VrpnDevice::trackerPosQuatCallback(void *userdata,
    const vrpn_TRACKERCB trackerData) {
  VrpnDevice* instance = (VrpnDevice*)userdata;
  if (instance) {
    instance->updateTracker(trackerData);
  } else {
    printd(WARNING,
        "VrpnDevice::trackerPosQuatCallback(): callback for unknown VRPN-device
            found!\n");
  } // else
} // trackerPosQuatCallback

void VRPN_CALLBACK VrpnDevice::buttonCallback(void *userdata, const vrpn_BUTTONCB
    buttonData) {
  VrpnDevice* instance = (VrpnDevice*)userdata;
  if (instance) {
    instance->updateButton(buttonData);
  } else {
    printd(WARNING,
        "VrpnDevice::buttonCallback(): callback for unknown VRPN-device found!\n");
  } // else
} // buttonCallback

void VRPN_CALLBACK VrpnDevice::analogCallback(void *userdata, const vrpn_ANALOGCB
    analogData) {
  VrpnDevice* instance = (VrpnDevice*)userdata;
  if (instance) {
    instance->updateAnalog(analogData);
  } else {
    printd(WARNING,
        "VrpnDevice::analogCallback(): callback for unknown VRPN-device found!\n");
  } // else
} // analogCallback
```

Listing 4.4: VrpnDevice.cpp - static VRPN callback methods

The implementation of the separate update methods is also quite similar. In each method the index of the corresponding button, axis or sensor is checked and if the index is in a valid range the values stored in the according vectors are updated. Note that before and after each update of these vectors the `acquireControllerLock`() and `releaseControllerLock`() methods are called. This is needed in order to avoid the simultaneous reading and writing of input values (e.g. from different threads). When updating the input values inside the `update`() method this lock is acquired automatically.
Finally the method prints a warning message once in case the obtained index for the button, axis or sensor is out of the provided range (this is why the `...CallbackWarnings` members are needed).
One difference in the `updateButton`() method in comparison to the other methods is that the `sendButtonChangeNotification`() method is called additionally in case the state of a button has changed. This call is needed by the abstract *inVRs* `Controller` in order to notify all registered listeners that a button value has changed.

```
void VrpnDevice::updateTracker(const vrpn_TRACKERCB trackerData) {
  int sensorIndex = trackerData.sensor - 1;
  if (sensorIndex < 0)
    return;

  acquireControllerLock();
  if (sensorIndex < (int)sensorValues.size()) {
    sensorValues[sensorIndex].position = gmtl::Vec3f(trackerData.pos[0],
        trackerData.pos[1],
        trackerData.pos[2]);
    sensorValues[sensorIndex].orientation = gmtl::Quatf(trackerData.quat[0],
        trackerData.quat[1],
        trackerData.quat[2], trackerData.quat[3]);
  } // if
  else if (sensorCallbackWarnings.find(sensorIndex) == sensorCallbackWarnings.end()
      ) {
    printd(WARNING,
        "VrpnDevice::updateTracker(): invalid tracker with index %i found - device
            is configured for only %i sensors! Further warnings for this sensor
            will be omitted!\n",
        sensorIndex, sensorValues.size());
    sensorCallbackWarnings.insert(sensorIndex);
  } // else
  releaseControllerLock();
} // updateTracker

void VrpnDevice::updateButton(const vrpn_BUTTONCB buttonData) {
  int buttonIndex = buttonData.button;
  int buttonValue = buttonData.state ? 1 : 0;
  bool change = false;

  acquireControllerLock();
  if (buttonIndex < (int)buttonValues.size()) {
    if (buttonValues[buttonIndex] != buttonValue) {
      buttonValues[buttonIndex] = buttonValue;
      change = true;
    } // if
  } // if
  else if (buttonCallbackWarnings.find(buttonIndex) == buttonCallbackWarnings.end()
      ){
    printd(WARNING,
        "VrpnDevice::updateButton(): invalid button with index %i found - device is
             configured for only %i buttons! Further warnings for this button will
            be omitted!\n",
        buttonIndex, buttonValues.size());
    buttonCallbackWarnings.insert(buttonIndex);
  } // else
```

```
  releaseControllerLock();

  if (change)
    sendButtonChangeNotification(buttonIndex, buttonValue);

  if (buttonIndex < (int)buttonValues.size()) {
    printd(INFO, "VrpnDevice::updateButton(): updated value of button %i: %i!\n",
        buttonIndex,
        buttonData.state);
  } // if
} // updateButton

void VrpnDevice::updateAnalog(const vrpn_ANALOGCB analogData) {
  int numAxes = analogData.num_channel;

  acquireControllerLock();
  for (int i=0; i < numAxes; i++) {
    if (i < (int)axisValues.size()) {
      axisValues[i] = analogData.channel[i];
      printd(INFO, "\taxis %i: %f\n", i, axisValues[i]);
    } // else if
    else if (axisCallbackWarnings.find(i) == axisCallbackWarnings.end()){
      printd(WARNING,
          "VrpnDevice::updateAnalog(): invalid axis with index %i found - device is
               configured for only %i axes! Further warnings for this axis will be
              omitted!\n",
          i, axisValues.size());
      axisCallbackWarnings.insert(i);
    } // else
  } // for
  releaseControllerLock();
} // updateAnalog
```

Listing 4.5: VrpnDevice.cpp - update methods

Now that the callback mechanism is described the only thing which still has to be called is the `mainloop()` method of the VRPN objects. These methods are called in the update method of the `VrpnDevice`.

```
void VrpnDevice::update() {
  if (tracker)
    tracker->mainloop();
  if (button)
    button->mainloop();
  if (analog)
    analog->mainloop();
} // update
```

Listing 4.6: VrpnDevice.cpp - update()

This is all what is needed in order to get the input values from the VRPN library into the `VrpnDevice` class. For publishing these values to the *inVRs* `Controller` the virtual methods of the `InputDevice` class have to be implemented:

```
int VrpnDevice::getNumberOfButtons() {
  return buttonValues.size();
} // getNumberOfButtons

int VrpnDevice::getNumberOfAxes() {
  return axisValues.size();
} // getNumberOfAxes

int VrpnDevice::getNumberOfSensors() {
  return sensorValues.size();
```

```
} // getNumberOfSensors

int VrpnDevice::getButtonValue(int idx) {
  int result = 0;
  if (idx >= 0 && idx < (int)buttonValues.size()) {
    result = buttonValues[idx];
  } // if
  else {
    printd(WARNING,
        "VrpnDevice::getButtonValue(): invalid button index %i passed (device has %
            i buttons)!\n",
        idx, buttonValues.size());
  } // if
  return result;
} // getButtonValue

float VrpnDevice::getAxisValue(int idx) {
  float result = 0;
  if (idx >= 0 && idx < (int)axisValues.size()) {
    result = axisValues[idx];
  } // if
  else {
    printd(WARNING,
        "VrpnDevice::getAxisValue(): invalid axis index %i passed (device has %i
            axes)!\n",
        idx, axisValues.size());
  } // if
  return result;
} // getAxisValue

SensorData VrpnDevice::getSensorValue(int idx) {
  SensorData result = IdentitySensorData;
  if (idx >= 0 && idx < (int)sensorValues.size()) {
    result = sensorValues[idx];
  } // if
  else {
    printd(WARNING,
        "VrpnDevice::getSensorValue(): invalid sensor index %i passed (device has %
            i sensors)!\n",
        idx, sensorValues.size());
  } // if
  return result;
} // getSensorValue
```

Listing 4.7: VrpnDevice.cpp - accessor methods for input data

Now with this methods the `Controller` can access the input data obtained from the VRPN library and can publish it to the *inVRs* application.
What still has to be done is the implementation of the `VrpnDeviceFactory` class. This class is used during loading of the `ControllerManager` configuration in order to create a `VrpnDevice` instance. The `VrpnDeviceFactory` must therefore provide a single method `create`() which takes two parameters: the first parameter defines the type of the `InputDevice` which should be created and the second parameter contains an `ArgumentVector` which is read from the configuration file. At first the method checks if the passed `className` matches to the class the factory can create (namely *VrpnDevice*). If not the method must return NULL, so that the `ControllerManager` knows that it has to call another factory class. If the class name matches then the method checks if an `ArgumentVector` was passed and if this parameter contains the *deviceID* argument. This argument is needed in order to find the VRPN device to which the connection should be established. If this check was successful then the method reads the VRPN device identifier and the number of buttons, sensors and axes (if defined) from the `ArgumentVector`. After having obtainted these values a new `VrpnDevice` object is created. Finally the method checks if the device could be initialized successfully and returns the device.

```
InputDevice* VrpnDeviceFactory::create(std::string className, ArgumentVector* args)
    {
  if (className != "VrpnDevice")
    return NULL;

  if (!args || !args->keyExists("deviceID")) {
    printd(ERROR,
        "VrpnDeviceFactory::create(): missing argument entry deviceID! Cannot
            create Device!\n");
    return NULL;
  } // if

  std::string deviceId;
  unsigned numSensors = 0;
  unsigned numButtons = 0;
  unsigned numAxes = 0;
  args->get("deviceID", deviceId);
  if (args->keyExists("numSensors"))
    args->get("numSensors", numSensors);
  if (args->keyExists("numButtons"))
    args->get("numButtons", numButtons);
  if (args->keyExists("numAxes"))
    args->get("numAxes", numAxes);

  VrpnDevice* device = new VrpnDevice(deviceId, numSensors, numButtons, numAxes);

  // check if device could be initialized and return null if not!
  if (!device->isInitialized()) {
    printd(ERROR,
        "VrpnDeviceFactory::create(): unable to initialize VRPN device with ID %s\n
            ",
        deviceId.c_str());
    delete device;
    device = NULL;
  } // if

  return device;
} // create
```

Listing 4.8: VrpnDevice.cpp - VrpnDeviceFactory::create()

This is everything which has to be implemented in order to integrate the input data from a VRPN device into *inVRs*. In the next section the integration of this device into the tutorial application is described.

4.4 Interconnecting own Devices with inVRs

There are many ways to provide tracking information to the system. In the last section we have learned how create our own *inVRs* devices based on existing libraries like for example VRPN.
In this section we will have a look on how to integrate a self-developed device into an *inVRs* application. Therefore the class `VrpnDevice` which was described in the previous section will be integrated into the *Going Immersive* tutorial application. Besides the `VrpnDevice` *inVRs* also provides an implementation for a device using the trackD library, namely the `TrackdDevice`. In order to allow users of trackD to also use tracking in this tutorials the snippets in this section are designed in a way to support both devices.
If you don't have a tracking system available but want to test this application anyways you can skip the following steps and continue with the chapter 5. The tutorial application is configured by default to provide a tracking system emulation device which you can use for simulating the tracking input then.
But now let's start with the integration of the tracking devices. In order to be able to use non-default input devices in an application the `ControllerManager` must at first be aware of these

devices. This is achieved by registering the factories for these devices in the class. Therefore at first the header files for the input devices we want to add have to be included. The includes for the `VrpnDevice` and the `TrackdDevice` are surrounded by `#ifdef` statements which are needed to avoid the use of specific VRPN or trackD datatypes. The checked defines are set by CMake automatically at configuration time, the detailed functionality will be described at the end of this section.

```
#ifdef WITH_VRPN_SUPPORT
#include <inVRs/tools/libraries/VrpnDevice/VrpnDevice.h>
#endif
#ifdef WITH_TRACKD_SUPPORT
#include <inVRs/tools/libraries/TrackdDevice/TrackdDevice.h>
#endif
```

Listing 4.9: CodeSnippets2.cpp - Snippet-2-1 → GoingImmersive.cpp

After the header files are included the new devices can be registered at the `ControllerManager`. This has to be done before the `ControllerManager` is configured in order to allow to create instances of the new devices as soon as the configuration file is loaded. Therefore the registration is implemented in the virtual `initInputInterfaceCallback()` method which is provided by the `OpenSGApplicationBase`. Again the checks for the defines are included to avoid the use of libraries which are not installed on your system.

```
  void initInputInterfaceCallback(ModuleInterface* moduleInterface) {
#ifdef WITH_VRPN_SUPPORT
    if (moduleInterface->getName() == "ControllerManager") {
      ControllerManager* contInt = dynamic_cast<ControllerManager*>(moduleInterface
          );
      assert(contInt);
      contInt->registerInputDeviceFactory(new VrpnDeviceFactory);
    } // if
#endif
#ifdef WITH_TRACKD_SUPPORT
    if (moduleInterface->getName() == "ControllerManager") {
      ControllerManager* contInt = dynamic_cast<ControllerManager*>(moduleInterface
          );
      assert(contInt);
      contInt->registerInputDeviceFactory(new TrackdDeviceFactory);
    } // if
#endif
  } // initInterfaceCallback
```

Listing 4.10: CodeSnippets2.cpp - Snippet-2-2 → GoingImmersive.cpp

Now that the factories are registered the `ControllerManager` is able to create `TrackdDevices` and `VrpnDevices` if configured in the configuration file. Next the configurations for the abstract *inVRs* controller has to be created. For the sake of simplicity two configuration files are already contained in this tutorial, one which uses a single VRPN device and another one for using a single trackD device for input. In the following the configuration for the VRPN device is presented, the file for trackD is nearly identical and therefore not described here.
The configuration file `VrpnController.xml` defines a `Controller` which consists of a single device of the type `VrpnDevice`. The argument *deviceID* defines the VRPN device identifier which is used in order to connect to the VRPN library. Additionally this device is configured to provide 3 buttons, 2 axes, and 2 sensors to the controller. Afterwards the mapping of the device buttons, axes and sensors to the controller values is done. In this case the indices of the VRPN device are equal to the ones used in the `Controller`. Since no other input device than the `VrpnDevice` is used the controller has the same number of buttons, axes and sensors (it could also have less, if not all values are mapped from the `VrpnDevice` to the controller). A more detailed description for the

ControllerManager configuration is given by the *Configuring the inVRs framework* document.

```
<?xml version="1.0"?>
<!DOCTYPE controllerManager SYSTEM "http://dtd.inVRs.org/controllerManager_v1.0a4.
    dtd">
<controllerManager version="1.0a4">
  <controller buttons="3" axes="2" sensors="2">
    <device type="VrpnDevice">
      <arguments>
        <arg key="deviceID" type="string" value="tracker@127.0.0.1"/>
        <arg key="numButtons" type="uint" value="3"/>
        <arg key="numAxes" type="uint" value="2"/>
        <arg key="numSensors" type="uint" value="2"/>
      </arguments>
      <button deviceIndex="0" controllerIndex="0"/>
      <button deviceIndex="1" controllerIndex="1"/>
      <button deviceIndex="2" controllerIndex="2"/>
      <axis deviceIndex="0" controllerIndex="0">
        <axisCorrection scale="1" offset="0"/>
      </axis>
      <axis deviceIndex="1" controllerIndex="1">
        <axisCorrection scale="1" offset="0"/>
      </axis>
      <sensor deviceIndex="0" controllerIndex="0">
        <coordinateSystemCorrection>
          <translation x="0" y="0" z="0"/>
          <rotation x="1" y="0" z="0" angleDeg="0"/>
          <scale x="1" y="1" z="1"/>
        </coordinateSystemCorrection>
      </sensor>
      <sensor deviceIndex="1" controllerIndex="1">
        <coordinateSystemCorrection>
          <translation x="0" y="0" z="0"/>
          <rotation x="1" y="0" z="0" angleDeg="0"/>
          <scale x="1" y="1" z="1"/>
        </coordinateSystemCorrection>
      </sensor>
    </device>
  </controller>
</controllerManager>
```

Listing 4.11: VrpnController.xml

In order to use this ControllerManager configuration file instead of the default one we have to change the entry in the InputInterface configuration file `inputinterface.xml` with the Snippet 2-1. Take care to remove or comment out the line above the snippet.

```
<!-- IMPORTANT: replace line above with this snippet!                    -->
  <module name="ControllerManager" configFile="VrpnController.xml"/>
```

Listing 4.12: XmlSnippets2.xml - Snippet2-1 → inputInterface.xml

Now the ControllerManager tries at startup to load the Controller which is defined in the `VrpnController.xml` file.
In order to use the Controller effectively also the Navigation configuration should be changed. Previously the Navigation was configured to work with a keyboard and a mouse. The new Controller configuration is now similar to a wand device, which has two axes and three buttons. Thus we will change the Navigation configuration to work with these input values instead. The configuration is already provided in the tutorial and can be found in the file `wandNavigation.xml`. The configuration file defines three models, the *translationModel* which describes the linear movement direction, the *speedModel* which defines the speed of the linear motion and the *orientationModel* which defines the change of orientation. For determining the linear movement direction the

`TranslationViewDirectionModel` is used which always returns the view direction of the camera. For the linear speed calculation the `SpeedButtonModel` is used which defines two buttons of the `Controller` which are used for forward and backward movement. And finally for determination of the orientation change the `OrientationDualAxisModel` is used which uses two axes for changing the rotation along two axes (X and Y) and an additional button for switching to the third rotation axis (Z).

```
<?xml version="1.0"?>
<!DOCTYPE navigation SYSTEM "http://dtd.inVRs.org/navigation_v1.0a4.dtd">
<navigation version="1.0a4">
  <translationModel type="TranslationViewDirectionModel"/>
  <orientationModel type="OrientationDualAxisModel" angle="10">
    <arguments>
      <arg key="xAxisIndex" type="int" value="0"/>
      <arg key="yAxisIndex" type="int" value="1"/>
      <arg key="buttonIndex" type="int" value="2"/>
    </arguments>
  </orientationModel>
  <speedModel type="SpeedButtonModel" speed="5">
    <arguments>
      <arg key="accelButtonIndex" type="int" value="0"/>
      <arg key="decelButtonIndex" type="int" value="1"/>
    </arguments>
  </speedModel>
</navigation>
```

Listing 4.13: wandNavigation.xml

In order to use this `Navigation` model the configuration file has to be exchanged in the file `modules.xml`:

```
<!-- IMPORTANT: replace line above with this snippet!                    -->
  <module name="Navigation" configFile="wandNavigation.xml"/>
```

Listing 4.14: XmlSnippets2.xml - Snippet2-2 → modules.xml

Before rebuilding and starting the application now you should check if your *inVRs* installation and the tutorial was build with VRPN and/or trackD support. For your *inVRs* installation you can simply look at the *inVRs* library directory and search for the according libraries (e.g. for VRPN `libinVRsVrpnDevice.so` on Linux, or `inVRsVrpnDevice.dll` on Windows, or `libinVRsVrpnDevice.dylib` on Mac OS X). If the libraries are not available you may have to rebuild *inVRs* and activate the VRPN or trackD support in the CMake GUI. The same has to be done for the CMake configuration of the *Going Immersive* tutorial. Details on the installation can be found in the appendix.
When you start the application now you should be able to use the axes and buttons of your VRPN or trackD device for navigation. The tracking information is not used yet, but will be used in the following chapters.

4.5 Summary

This chapter has briefly introduced different VR input devices and shown how to interconnect them with the *inVRs* framework. An abstract controller which maps the physical devices on abstract *inVRs* data has been described and the configuration of the controller has been explained in detail. Often own libraries are used to access input devices. Thus the implementation for a binding to the abstract controller has been explained. VRPN was used as a demonstrator for this binding.

The reader should now be able to develop own device bindings to the *inVRs* framework by implementing a class derived from the InputDevice.

Chapter 5

Working with Avatars

In the previous chapter we have seen how to use own devices and incorporate tracking systems in order to navigate through the environment. The gathered tracking data can be used for example for interaction purposes.
The physical world position and orientation data gathered by the tracking system is not restricted to be used only for interaction tasks. It can be incorporated as well for the display of remote users. We will have a closer look at the user representation. This representation of a user in a VE is commonly known as an avatar.
In the *Medieval Town Tutorial* we have only used static avatars represented by a simple model. These basic avatars are implemented in the class `SimpleAvatar`. A typical static avatar is described in *inVRs* by a configuration file as given below.

```
<?xml version="1.0"?>
<!DOCTYPE simpleAvatar SYSTEM "http://dtd.inVRs.org/simpleAvatar_v1.0a4.dtd">
<simpleAvatar version="1.0a4">
  <name value="MedievalCitizen"/>
  <representation>
    <file type="VRML" name="undead.wrl"/>
    <transformation>
      <translation x="0" y="0" z="0"/>
      <rotation x="0" y="1" z="0" angleDeg="180"/>
      <scale x="0.08" y="0.08" z="0.08"/>
    </transformation>
  </representation>
</simpleAvatar>
```

Listing 5.1: simpleAvatar.xml

The model which is to be loaded is identified by the <file>-node, while an additional transformation can be applied by the <transformation>-node.
Other types of avatars are available as well, which can make use of the data gathered by tracking systems in order to provide a visual approximation of the actual users pose.

5.1 Modelling and Exporting Avatars

In general you need a modeling tool for creating *inVRs* avatars. Simple avatars as described in the previous section can be easily exported from any modeling tool, which supports the file formats readable by *inVRs* and resprectively the underlying scene graph used.
The advanced Avatara avatars make use of so called mesh skinning. Basically a geometry is interconnected with a set of axes representing the bones of the avatar. By using these avatars it is possible to define animation sequences. It gives as well individual access to the head bone, the spine bone and the hand position and orientation.

The *inVRs* avatars can be modeled with a variety of modelling tools. Additional scripts and tools for the Avatara package exist for exporting the avatars into a format usable by *inVRs*. For a detailed instruction how to model and export avatars please refer to the *Avatara Manual*. Three different modeling tools are currently supported for the export of the avatars. 3D Studio Max [1], MAYA [2] and Blender [3] export the avatar models including animation phases.

5.2 Using Avatara

The *inVRs* framework provides an external package called Avatara, which was developed by Helmut and Martin Garstenauer. Avatara is implemented as a scene graph specific tool for OpenSG. The can be attached as a simple OpenSG node. A detailed description on Avatara avatars is given in the *Avatara Manual*.
The avatars of the Avatara package offer basically three different functionalities:

- Starting and stopping of animation phases
- Moving the head bone
- Setting the hand position and orientation

If we take a look at the configuration of the Avatara avatars we can see a significant difference to our previously used simple avatars.

```
<?xml version="1.0"?>
<!DOCTYPE simpleAvatar SYSTEM "http://dtd.inVRs.org/avataraAvatar_v1.0a4.dtd">
<avataraAvatar version="1.0a4">
  <name value="Undead"/>
  <representation>
    <file type="Avatara MDL" name="undead/undead.mdl"/>
    <transformation>
      <translation x="0" y="0" z="0"/>
      <rotation x="1" y="0" z="0" angleDeg="-90"/>
      <scale x="-0.08" y="-0.08" z="0.08"/>
    </transformation>
  </representation>
  <texture file="undead/undeadN.jpg"/>
  <animations smooth="true" speed="2" default="standing">
    <animation name="standing" file="undead/undead_standing.ani"/>
    <animation name="walking" file="undead/undead_walking.ani"/>
  </animations>
</avataraAvatar>
```

Listing 5.2: undead.xml

An additional <animations>-node provides a list of animation sequences which can be played on demand. It can be defined whether the transition between the sequences is to be smooth which is achieved by interpolating between the different stages. The speed of the animation sequences can be set and a default animation sequence can be set.
Additionally the format of the model and the animation sequences to be loaded is proprietary and can be exported by the described modeling tools.

5.3 Integrating an Avatara Avatar into the tutorial

Usually an avatar is used in an application to represent the user in the virtual world. Therefore the avatar is displayed at the position the user has currently navigated to. In this tutorial we

[1] http://www.autodesk.de/adsk/servlet/index?siteID=403786&id=10612077
[2] http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=7635018
[3] http://www.blender.org

will change this default behaviour a bit. The avatar in this tutorial will be used to display the transformation of the user as it is determined by the tracking system. This will allow us to interpret the values provided by the tracking system in a graphical way.
At first the `WorldDatabase` needs a factory for creating an `AvataraAvatar` at configuration loading time. Thus we have to register this factory in our application. The header file containing the `AvataraAvatarFactory` has to be included first:

```
#include <inVRs/tools/libraries/AvataraWrapper/AvataraAvatar.h>
```

Listing 5.3: CodeSnippets3.cpp - Snippet-3-1 → GoingImmersive.cpp

Next the factory is registered in the `WorldDatabase`. This is done in the `initCoreComponent()` method which is called before the components of the `SystemCore` are configured. The method checks for the callback for the `UserDatabase` initialization. This is done because the avatar configuration is referenced by the `UserDatabase` configuration, although the `WorldDatabase` is responsible for loading the avatar. When the callback for the `UserDatabase` initialization is executed the avatar factory is registered in the `WorldDatabase`.

```
void initCoreComponentCallback(CoreComponents comp) {
  // register factory for avatara avatars
  if (comp == USERDATABASE) {
    WorldDatabase::registerAvatarFactory (new AvataraAvatarFactory());
  } // else if
} // initCoreComponentCallback
```

Listing 5.4: CodeSnippets3.cpp - Snippet-3-2 → GoingImmersive.cpp

Now that the avatara avatar can be loaded the `UserDatabase` configuration file must be updated in order to define the configuration file for the avatar.

```
<avatar configFile="undead.xml"/>
```

Listing 5.5: XmlSnippets3.xml - Snippet 3-1 → userDatabase.xml

This is all that has to be done in order to get the avatar into the application. What is still missing is the update of the avatar's transformation. Usually this is done by the `TransformationManager` when updating the navigated transformation of the user. But in this tutorial the avatar should not represent the user in the virtual world but visualize the user as recognized by the tracking system. Thus we have to update the avatar transformation manually in our application. For this we need at first two variables, one for storing the pointer to the avatar and another one for defining the initial transformation of the avatar in the scene. This initial transformation represents the center point of the tracking system (0,0,0) in the virtual world.

```
AvatarInterface* avatar;
gmtl::Vec3f COORDINATE_SYSTEM_CENTER;
```

Listing 5.6: CodeSnippets3.cpp - Snippet-3-3 → GoingImmersive.cpp

In the constructor of the application these two variables are then initialized. The initial transformation of the avatar is set to the center point of the platform in the middle of the scene.

```
  avatar = NULL;
  COORDINATE_SYSTEM_CENTER = gmtl::Vec3f(5, 1, 5);
```

Listing 5.7: CodeSnippets3.cpp - Snippet-3-4 → GoingImmersive.cpp

In the `initialize()` the pointer to the avatar is now obtained from the local `User` object. The variable `localUser` which is used therefore is provided by the `OpenSGApplicationBase`.

```
    avatar = localUser->getAvatar();
    if (!avatar) {
      printd(ERROR,
          "GoingImmersive::initialize(): unable to obtain avatar! Check
              UserDatabase configuration!\n");
      return false;
    } // if
```

Listing 5.8: CodeSnippets3.cpp - Snippet-3-5 → GoingImmersive.cpp

Now that the pointer to the avatar is obtained we can write a method which updates the transformation of the avatar. This method first requests the transformation of the user relative to the tracking system center. By default a tracking system has at least two sensors, one for the head transformation of the user and another one for the transformation of the user's hand (or input device). This allows for correcting the perspective for this user and enables interaction but does not provide the correct position of the user (meaning the point where the user's feet hit the ground) relative to the tracking system center. To get this position exactly an additional sensor would be needed at the feet of the user. To avoid this additional sensor *inVRs* provides a concept which let's you calculate the user transformation relative to the tracking system center, called the `UserTransformationModel`. Several implementations can exist for this model, by default *inVRs* uses the `HeadPositionUserTransformationModel`. This model takes the position of the sensor used for head tracking (which has index 0 by default) and removes the height value to approximate the user position. This transformation is also the one which is requested here in the application. This transformation is then added to the center transformation of the platform and finally set as avatar transformation.

```
  void updateAvatar() {
    TransformationData trackedUserTrans = localUser->getTrackedUserTransformation()
        ;
    trackedUserTrans.position += COORDINATE_SYSTEM_CENTER;
    avatar->setTransformation(trackedUserTrans);
  } // updateAvatar
```

Listing 5.9: CodeSnippets3.cpp - Snippet-3-6 → GoingImmersive.cpp

In order to update this transformation continuously this method has to be called once a frame. This is achieved by adding a method call into the `update()` method.

```
    updateAvatar();
```

Listing 5.10: CodeSnippets3.cpp - Snippet-3-7 → GoingImmersive.cpp

Now the avatar is fully integrated into our application and is displayed at the transformation determined by the tracking system. When you start the application now you should see the movement of the avatar according to your movement tracked by the tracking system.

5.4 Testing the avatar without a tracking system

In case you don't have a tracking system available you can use an emulator device provided by *inVRs*, the `GlutSensorEmulatorDevice`. This device allows you to simulate the output of a tracking system with the help of a mouse and a keyboard. By default the *GoingImmersive* tutorial

application is configured to use this device for the abstract *inVRs* `Controller`. The device is configured in the `ControllerManager` configuration file `MouseKeybSensorController.xml`:

```
<?xml version="1.0"?>
<!DOCTYPE controllerManager SYSTEM "http://dtd.inVRs.org/controllerManager_v1.0a4.
    dtd">
<controllerManager version="1.0a4">
  <controller buttons="11" axes="2" sensors="2">
    <device type="GlutMouseDevice">
      <arguments>
        <arg key="axisReleaseSpeed" type="float" value="20"/>
      </arguments>
      <button deviceIndex="0" controllerIndex="0"/>
      <button deviceIndex="1" controllerIndex="1"/>
      <button deviceIndex="2" controllerIndex="2"/>
      <axis deviceIndex="0" controllerIndex="0">
        <axisCorrection scale="1" offset="0"/>
      </axis>
      <axis deviceIndex="1" controllerIndex="1"/>
    </device>
    <device type="GlutCharKeyboardDevice">
      <button deviceIndex="119" controllerIndex="3" /> <!-- W -->
      <button deviceIndex="115" controllerIndex="4" /> <!-- S -->
      <button deviceIndex="97"  controllerIndex="5" /> <!-- A -->
      <button deviceIndex="100" controllerIndex="6" /> <!-- D -->
      <button deviceIndex="56"  controllerIndex="7" /> <!-- keypad 8 -->
      <button deviceIndex="53"  controllerIndex="8" /> <!-- keypad 5 -->
      <button deviceIndex="52"  controllerIndex="9" /> <!-- keypad 4 -->
      <button deviceIndex="54"  controllerIndex="10"/> <!-- keypad 6 -->
    </device>
    <device type="GlutSensorEmulatorDevice">
      <arguments>
        <arg key="numberOfSensors" type="uint" value="2"/>
        <arg key="switchSensorButton" type="uint" value="256"/>
        <arg key="switchTransformationTargetButton" type="uint" value="257"/>
        <arg key="switchAxesButton" type="uint" value="258"/>
      </arguments>
      <sensor deviceIndex="0" controllerIndex="0">
        <coordinateSystemCorrection>
          <translation x="0" y="0" z="0"/>
          <scale x="100" y="100" z="100"/>
        </coordinateSystemCorrection>
      </sensor>
      <sensor deviceIndex="1" controllerIndex="1">
        <coordinateSystemCorrection>
          <translation x="0" y="0" z="0"/>
          <scale x="100" y="100" z="100"/>
        </coordinateSystemCorrection>
      </sensor>
    </device>
  </controller>
</controllerManager>
```

Listing 5.11: MouseKeybSensorController.xml

In this configuration file the `Controller` is composed out of three different input devices: the `GlutMouseDevice` which is used to get the input from a mouse, the `GlutKeyboardDevice` which reads the keyboard keys as buttons and the `GlutSensorEmulatorDevice` which creates virtual sensor values with the help of mouse and keyboard.
The first argument for the `GlutSensorEmulatorDevice` defines the number of emulated sensors which are provided by this device. In this case 2 sensors are simulated. The next argument defines which button is used to switch between the sensors. The button values 0-255 are reserved for the keyboard buttons (ascii values), the buttons 256-258 correspond to the mouse buttons left, middle and right. The third argument defines which button is used to switch between the translation and

rotation of the currently emulated sensor. And the last argument defines which button is used to switch the Y and the Z axis in the sensor simulation.
In the the two following <sensor> elements the mapping from the sensor index in the device to the sensor index of the controller is established. In this definition you can see that the sensor values are scaled by the factor 100 in each axis, just to match to the values expected by the application.
When starting the application you can now switch between the normal navigation and the sensor emulation mode by pressing the keyboard button *1*. When pressing the left mouse button you can switch between the two sensors you want to manipulate. Pressing the middle mouse button allows you to switch between the manipulation of the translation and the rotation part of the current sensor. And pressing the right mouse button allows you to switch between the manipulation along the Y or Z axis of the current sensor.

5.5 Summary

This chapter has briefly introduced the concepts of avatars in an *inVRs* virtual world. The use of the external Avatara avatars has been described in detail. These avatars have been interconnected and can be used to display animation sequences so far. Now we should be able to display a user using tracking systems inside an immersive VE.

Chapter 6

Coordinate Systems

Coordinate systems play an important role in VEs. They are used to display objects at certain positions with a given orientation or even more to establish relationships between objects. In *inVRs* a set of coordinate systems is used to represent the users' avatars.
In *inVRs* the same coordinate system as in OpenGL is used, which is a right handed coordinate system where the axes point into the following directions:

- X-axis: point to the right
- Y-axis: points to top
- Z-axis: towards the observer out of the screen

6.1 User Coordinates

For interaction, navigation or simply to display an avatar several coordinate systems are necessary in the *inVRs* framework. Figure 6.1 illustrates the most important *inVRs* coordinate systems.

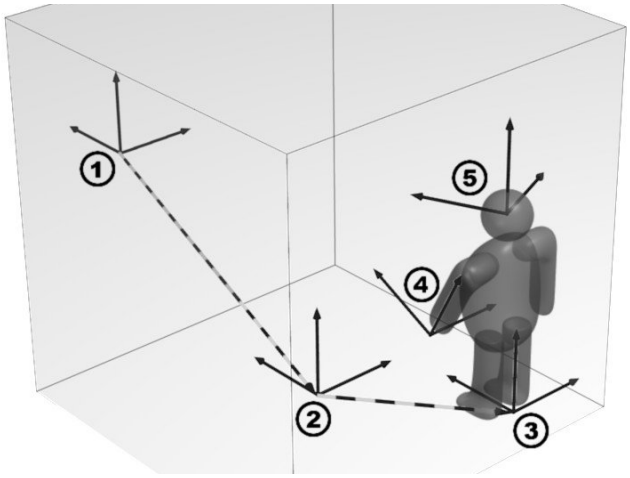

Figure 6.1: Coordinate Systems of the User

As illustrated in Figure 6.1 the following five coordinate systems are required for a correct user display and will be explained in the subsequent paragraphs:

1. Origin of the VE
2. Navigated transformation

3. User transformation
4. Hand transformation (often identical to the cursor transformation)
5. Head transformation (often identical to the camera transformation)

Navigated Transformation The navigated transformation marked by (2), is the result of navigation processing provided by the navigation module. This coordinate system of the navigated transformation is directly related to the origin of the VE. If no additional tracking information is provided the avatar is placed at the origin of this coordinate system. The navigated transformation is considered the origin of the tracking system if available.

User Transformation Besides on the navigated position of the user, other coordinate systems have to be set up to represent the embodiment of the user correctly in the NVE. When a tracking system is available the tracked position of the user can be added to the navigated position in order to determine the avatar transformation. By default the tracked position of the user is determined by taking the position of the head sensor and setting the height-value to 0 to approximate the position of the user's feet. If no tracking system is used the head tracking data is set to the identity matrix.
The user transformation can be obtained on two different ways, either by taking the transformation relative to the center of the tracking system which is called Tracked User Transformation in *inVRs* or relative to the origin of the VE, which is called World User Transformation. The world user transformation is therefore calculated by:

$$worldUserTrans = navigatedTrans * trackedUserTrans$$

Hand Transformation The hand transformation is the value provided by the hand sensor, which is typically integrated in the wand or attached to a data glove. By default the hand sensor in *inVRs* is the one with index 1 in the `Controller`. It provides information on where the avatar's hand should be located and can be used for the correct user representation display if inverse kinematics is used or for interaction purposes if the user's cursor is related directly to the hand.
The hand transformation can be provided either relative to the tracking system center, or relative to the user transformation or relative to the origin of the virtual world. The transformation relative to the tracking system center is directly provided by the sensor of the input device. Based on this the other transformations can be calculated by:

$$userHandTrans = trackedUserTrans^{-1} * trackedHandTrans$$
$$worldHandTrans = navigatedTrans * trackedHandTrans$$

Head Transformation The head transformation is provided by the head sensor gathered by the tracking system. By default the head sensor has the `Controller` index 0 in an *inVRs* application. The head transformation can be used for example to calculate the transformation of the camera. Like the hand transformation also the head transformation can be retrieved in three ways, either by directly from the sensor (which is relative to the tracking system center), or relative to the user transformation or relative to the virtual world center. Based on the first transformation the others can be calculated by:

$$userHeadTrans = trackedUserTrans^{-1} * trackedHeadTrans$$
$$worldHeadTrans = navigatedTrans * trackedHeadTrans$$

Sensor Transformations Besides the two special sensor transformations for head and hand the transformations of any number of additional sensors can be used in *inVRs*. These transformations can again be obtained directly from the tracking system, relative to the user or relative to the world:

$$i ... sensorIndex$$
$$userSensorTrans_i = trackedUserTrans^{-1} * trackedSensorTrans_i$$
$$worldSensorTrans_i = navigatedTrans * trackedSensorTrans_i$$

The special indices 0 and 1 return exactly the head and hand transformations described previously.

Cursor Transformation The cursor transformation is used to define the transformation of the virtual cursor relative to the origin of the virtual world. Different cursor transformation models are provided in *inVRs* for calculating this transformation. The information about the chosen cursor transformation model is stored in the user database.
The cursor of the user can be represented in many ways which will be explained in future tutorial.

Camera Transformation The transformation of the camera is calculated and constantly updated inside the `TransformationManager`. By default the camera transformation corresponds to the navigated transformation of the user.

6.2 Visualizing Transformations

In this section the current tutorial application is extended in order to display the transformations of the different sensors of the `Controller` in the virtual world. The transformations are visualized with the help of multiple entities which have a 3D model in the form of a coordinate system. One coordinate system entity will be used to visualize the tracked user transformation, the other entities visualize the tracked sensor transformations of each sensor provided by the `Controller`.
The first step in achieving this goal is to determine how many coordinate system entities have to be created in our application. One entity is already contained in the application and is displayed at the center of the platform. This one is will be used for visualizing the tracked user transformation. Thus the number of entities which still have to be created corresponds to the number of sensors provided by the `Controller`. Therefore at first a variable is defined in which the number of sensors is stored. The value for this variable is then obtained in the `initialize()` method.

```
int numberOfSensors;
```

Listing 6.1: CodeSnippets4.cpp - Snippet-4-1 → GoingImmersive.cpp

In the `initialize()` method at first the used controller is requested from the srcclassControllerManager. From this controller the number of sensors is then stored in the variable.
In the next step a new instance of the coordinate system entity is created for each sensor by calling the `createEntity()` method of the `WorldDatabase`. The first parameter of this method defines the ID of the `EntityType` from which an instance should be created. This ID is the one defined in the `EntityType` configuration file `entities.xml`. The second parameter defines the ID of the `Environment` in which the new `Entity` instance should be created. Our tutorial application only uses a single `Environment` with ID 1.

```
ControllerInterface* controller = controllerManager->getController();
if (!controller) {
  printd(ERROR,
      "GoingImmersive::initialize(): unable to obtain controller! Check
          ControllerManager configuration!\n");
```

```
    return false;
  } // if
  numberOfSensors = controller->getNumberOfSensors();

  // create an instance of the coordinate system entity (ID=10) for each
  // sensor in the environment with ID 1
  for (int i=0; i < numberOfSensors; i++) {
    WorldDatabase::createEntity(10, 1);
  } // for
```

Listing 6.2: CodeSnippets4.cpp - Snippet-4-2 → GoingImmersive.cpp

Now that all needed entities are created a method has to be added which updates the transformation of the coordinate system entities. This method will be called `updateCoordinateSystems()`. Inside this method at first the pointer to the `EntityType` which is used for the coordinate systems is obtained from the `WorldDatabase`. From this `EntityType` the list of all instances (coordinate system entities) is requested. Then the transformation of the first entity is calculated. This transformation represents the tracked user transformation, thus this transformation is requested from the local `User` object. To this transformation the position offset of the platform center is added (which represents to the tracking system center). Finally the `Entity` transformation is then updated with the calculated one.
After the tracked user transformation the tracked sensor transformations are updated. Therefore a loop iterates over the number of available sensors. Each single sensor transformation is then requested from the local `User` object. You may now wonder why these values are not taken from the `Controller` directly. In this application it would not make any change if the transformations would be obtained from the `Controller` instead. But using the methods from the `User` has two big benefits. The first one is that the transformation was pushed through a transformation pipe from the `TransformationManager`. Inside this pipe the sensor transformation could be smoothened or extrapolated for example, or another example would be to replay a recorded tracking data set via this pipe. The second benefit is that the tracked sensor transformations are also available for `User` objects from remote users, while the `Controller` only provides the local tracking data. Thus using the tracking data from the `User` object is always recommended.
After the tracked sensor transformation was obtained it is again added to the platform center and written to an coordinate system entity if available (what should be the case since we created them previously).

```
void updateCoordinateSystems() {
  TransformationData trackedUserTrans, sensorTrans;
  // get the list of coordinate system entities (entity type ID = 10)
  EntityType* coordinateSystemType = WorldDatabase::getEntityTypeWithId(10);
  const std::vector<Entity*>& entities = coordinateSystemType->getInstanceList();

  // map the tracked user transformation to the first entity
  if (entities.size() > 0) {
    trackedUserTrans = localUser->getTrackedUserTransformation();
    trackedUserTrans.position += COORDINATE_SYSTEM_CENTER;
    entities[0]->setEnvironmentTransformation(trackedUserTrans);
  } // if
  // map the tracked sensor transformations to the remaining entities
  for (int i=0; i < numberOfSensors; i++) {
    sensorTrans = localUser->getTrackedSensorTransformation(i);
    sensorTrans.position += COORDINATE_SYSTEM_CENTER;
    if (i+1 < entities.size()) {
      entities[i+1]->setEnvironmentTransformation(sensorTrans);
    } // if
  } // for
} // updateCoordinateSystems
```

Listing 6.3: CodeSnippets4.cpp - Snippet-4-3 → GoingImmersive.cpp

The last thing we still have to do is calling the update method continuously, thus adding the method call into the `update()` method.

```
    updateCoordinateSystems();
```

Listing 6.4: CodeSnippets4.cpp - Snippet-4-4 → GoingImmersive.cpp

When starting the application now you should be able to see the transformations of all sensors provided by your configured `Controller`.

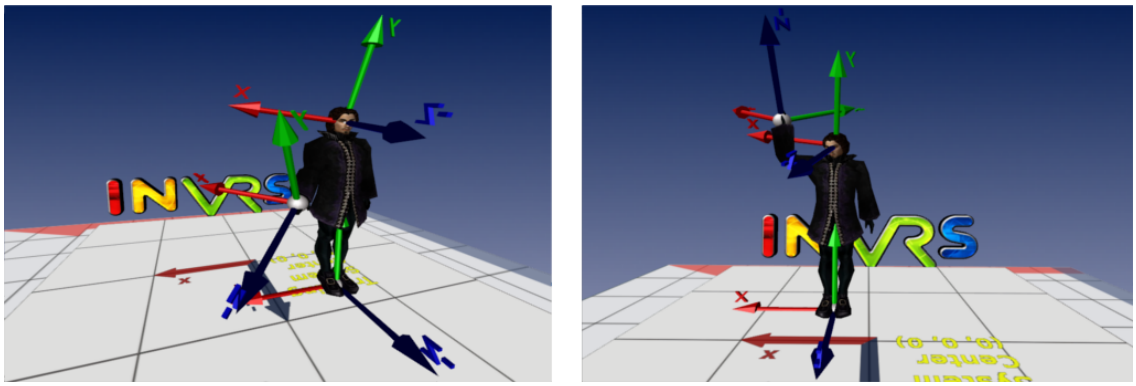

Figure 6.2: The Final Going Immersive Application

6.3 Summary

In this chapter an introduction into the *inVRs* user coordinates was given. The different coordinate systems used for the representation of the user as well as interaction purposes have been described. Additionally the visualisation of these coordinate systems has been performed. The reader should now be able to interact with VR input devices and multi-display setups. In combination with the *Medieval Town Tutorial* full NVEs with articulated avatars can be created.

Chapter 7

Outlook

This tutorial has shown how to use the *inVRs* framework with a variety of input devices an displays. At first wrapping functionality for faster application development was introduced. Basic setup as described in the *Medieval Town Tutorial* was hidden inside a so called application base. The use as well as the configuration of the CAVE Scene Manager was explained to ease the access to the OpenSG multi-display functionality. Afterwards the interconnection with arbitrary input devices was illustrated. It was shown how to connect your own input devices with *inVRs*. As an example an interconnection to VRPN as a rather generic input library was developed. In order to display remote users correctly the concept of articulated avatars was introduced. These avatars provided by the external Avatara package make use of tracking data to display head orientation and hand position and orientation correctly. When displaying a user in an NVE many different coordinate systems have to be configured which was shown in the end of the document.
The reader of this tutorial should now be able to interconnect arbitrary multi-display systems and input devices in order to create fully immersive NVEs. Interaction and navigation can be used as known from a previous tutorial. The introduced avatars can of course be integrated as well into simple desktop applications. By simply changing the setup of the CAVE Scene Manager setup and altering the used controller it is now easily possible to switch from desktop application with a mouse keyboard controll to installations likes CAVEs without altering a single line of code or recompiling.
All available OpenSG specific tools can be used in conjunction with the CAVE Scene Manager in order to provide stereoscopic output. When writing own OpenSG code an using the multi-display functionality one has to be very careful with node locking as in general with OpenSG multi-display applications. If the locking is not performed correctly the application might run on stable a single display system and might crash afterwards when multi-display output is configured.

7.1 Funky Physics

In order to generate really vivid and lifelike VEs often physics simulation is incorporated. The next tutorial will give an insight on how physics simulation can be done in *inVRs*.
This tutorial focuses on the *inVRs* physics module which is based on the Object Oriented Physics Simulation (OOPS) developed by Roland Landertshamer as an implementation basis for his MSc Thesis [Lan09]. The basics of rigid body dynamics will be introduced and the configuration of physically simulated entities will be explained in depth in the usual hands-on way.

7.2 Acknowledgments

The authors of the framework would like to thank the contributors of the core code, the tools as well as people who helped administrating the project for their selfless efforts and achievements. We would also like to thank all the users supporting us and evaluating the framework.

Considering the tools introduced in this tutorial special thanks go to their developers Adrian Haffegee, Helmut Garstenauer and Martin Garstenauer. Thanks so much.

Bibliography

[Abe04] Oliver Abert. *OpenSG Tutorial*, 2004.

[AV06] Christoph Anthes and Jens Volkert. invrs - a framework for building interactive networked virtual reality systems. In Michael Gerndt and Dieter Kranzlmüller, editors, *International Conference on High Performance Computing and Communications (HPCC '06)*, volume 4208 of *Lecture Notes in Computer Science (LNCS)*, pages 894–904, Munich, Germany, September 2006. Springer.

[CNSD+92] Carolina Cruz-Neira, Daniel J. Sandin, Thomas A. Defanti, Robert V. Kenyon, and John C. Hart. The cave: Audio visual experience automatic virtual environment. *Communications of the ACM*, 35(6):64–72, June 1992.

[CPS+97] Marek Czernuszenko, Dave Pape, Daniel J. Sandin, Thomas A. DeFanti, Gregory L. Dawe, and Maxine D. Brown. The immersadesk and infinity wall projection-based virtual reality displays. *Computer Graphics*, 31(2):46–49, May 1997.

[Haf04] Adrian Haffegee. Development of a scalable network topology supporting close-coupled collaboration. Master's thesis, University of Reading, UK, Reading, UK, 2004.

[HJAA05] Adrian Haffegee, Ronan Jamieson, Christoph Anthes, and Vassil N. Alexandrov. Tools for collaborative vr application development. In Vaidy S. Sunderam, Geert Dick van Albada, Peter M. A. Sloot, and Jack J. Dongarra, editors, *International Conference on Computational Science (ICCS '05)*, volume 3516 of *Lecture Notes in Computer Science (LNCS)*, pages 350–358, Atlanta, GA, USA, May 2005. Springer.

[KF94] Wolfgang Krueger and Bernd Fröhlich. The responsive workbench. *IEEE Computer Graphics and Applications*, 14(3):12–15, 1994.

[Lan09] Roland Landertshamer. Physics simulation in networked virtual environments. Master's thesis, Institut für Graphische und Parallele Datenverarbeitung, Johannes Kepler University Linz, Linz, Austria, August 2009.

[Rei02] Dirk Reiners. *OpenSG: A Scene Graph System for Flexible and Efficient Realtime Rendering for Virtual and Augmented Reality Applications*. PhD thesis, Technische Universität Darmstadt, Mai 2002.

[RS01] Gerhard Reitmayr and Dieter Schmalstieg. An open software architecture for virtual reality interaction. In *ACM Symposium on Virtual Reality Software and Technology (VRST '01)*, pages 47–54, Alberta, Canada, November 2001. ACM Press.

[RS05] Gerhard Reitmayr and Dieter Schmalstieg. Opentracker: A flexible software design for three-dimensional interaction. *Virtual Reality*, 9(1):79–92, December 2005.

[Sut68] Ivan E. Sutherland. A head-mounted three-dimensional display. In *Fall Joint Computer Conference AFIPS Conference*, pages 757–764, Fall 1968.

[THS+01] Russell M. Taylor II, Thomas C. Hudson, Adam Seeger, Hans Weber, Jeffrey Juliano, and Aron T. Helser. Vrpn: A device-independent, network-transparent vr peripheral system. In *ACM Symposium on Virtual Reality Software and Technology (VRST '01)*, pages 55–61, Alberta, Canada, November 2001. ACM Press.

List of Figures

Listings

Appendix

Used Models

Author: TiZeta
Model: Low Poly Undead Male Model
Avatar: http://e2-productions.com/

Installation Instructions

Prerequisites

To be able to build the tutorial application the following packages have to be installed:

1. *inVRs*: version 1.0alpha5 or above
2. *CMake*: version 2.6.x

Step 1: Creating Build Structure with CMake GUI

Before the tutorial application can be built the build files for your target system have to be created with the help of CMake. In this step we assume that you have graphical client for running CMake. If this is not the case and you have to run CMake from command line please follow the instructions in section *Step 1 (Alternative): Using CMake from Command Line*.
At first you have to open CMake and set the location of the tutorial sources and the directory where the build files should be stored. Afterwards press the *Configure* button:

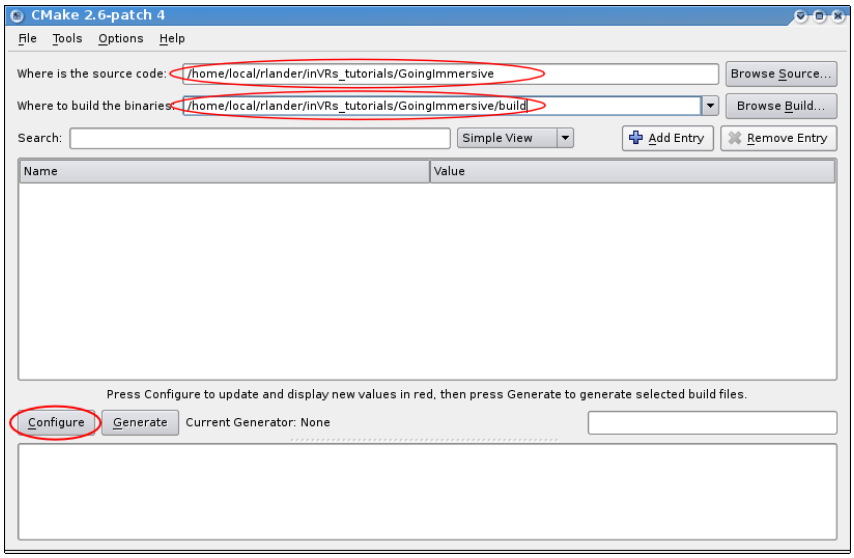

Figure 7.1: Open CMake and set GoingImmersive paths

Next you have to choose the target build structure. On Linux or Mac OS X systems for example you may want to select *Unix Makefiles* here, on Windows you should select your *Microsoft Visual Studio* version. Of course you can also select other target build structures, like *CodeBlocks* or *Xcode*. Next press the *Finish* button:

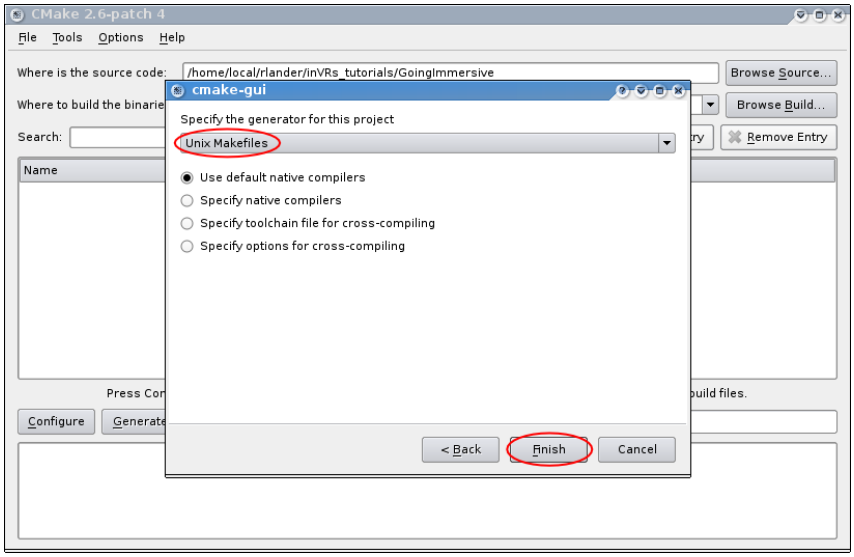

Figure 7.2: Open CMake and set GoingImmersive paths

Now CMake tries to find the paths to all libraries and include files needed by the tutorial application. Depending on your installation CMake may fail in this step, which results in a picture similar to the following:

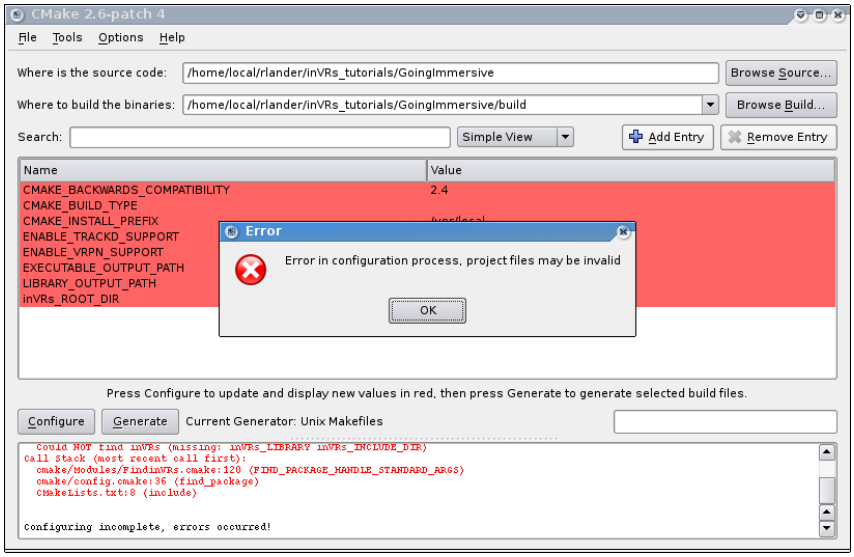

Figure 7.3: CMake failure when a path could not be found

Now after you pressed *OK* here you should be able to see the reason for the failure by reading the log output in the textfield at the bottom of the window. In this case the path to the *inVRs* installation was not found.

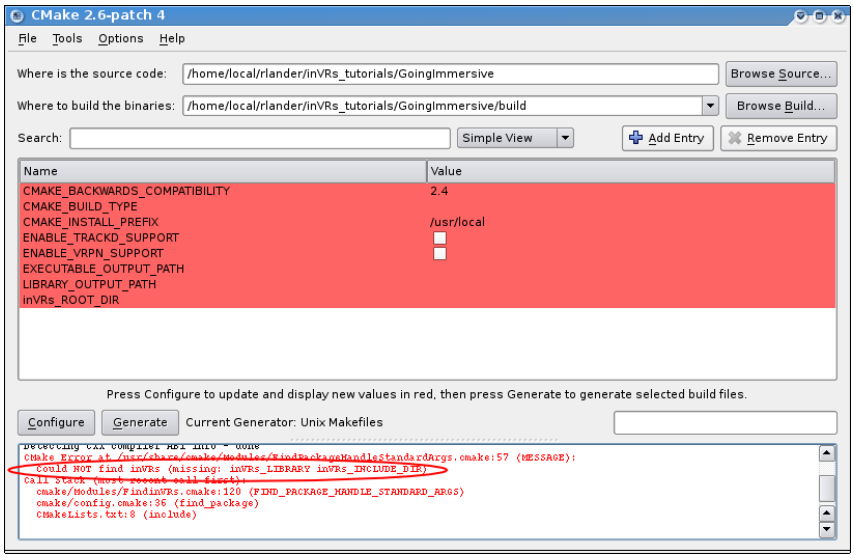

Figure 7.4: Reason for CMake failure in log output

Therefore you have to set the *inVRs_ROOT_DIR* variable to the folder of your *inVRs* installation and press *Configure* again.
Proceed the same way with other paths which can not be found until the configuration step finishes successfully.
In the next step you should choose whether you want to add a support for the VRPN and/or TrackD tracking library to your build system. Activating this support requires that also your

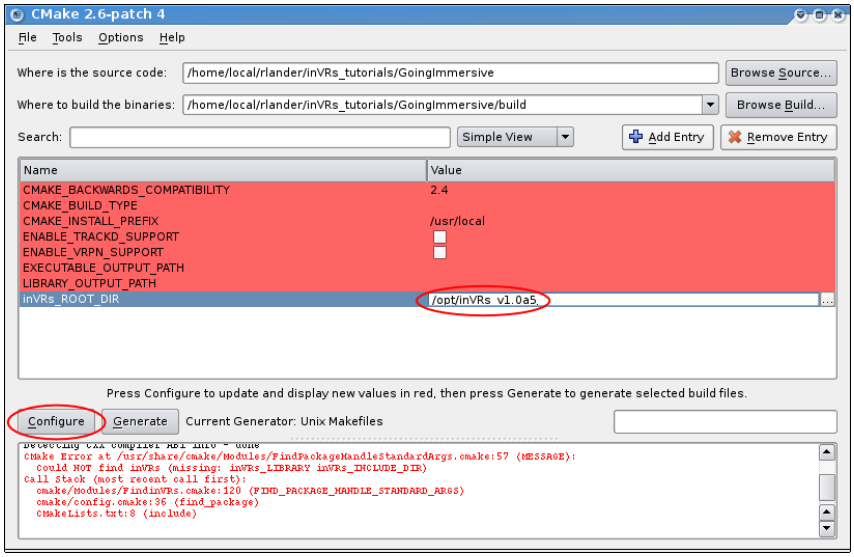

Figure 7.5: Enter missing path and continue

inVRs installation was built with the same libraries enabled and the according libraries must be installed on your system. If you want to run this tutorial without a tracking system you can ignore this step. In the following picture the support for VRPN will be activated and confirmed by pressing the *Configure* button again:

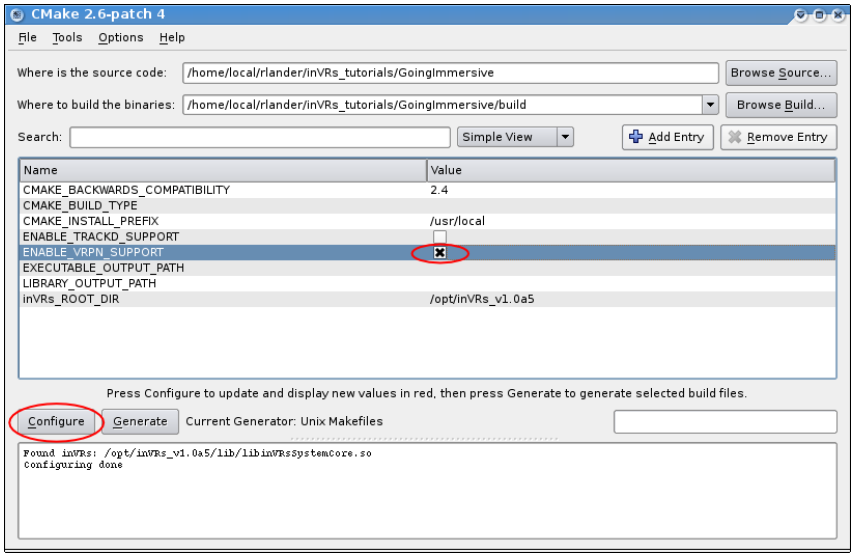

Figure 7.6: Activate VRPN-support for tutorial

Finally after the configuration was finished you can press the *Generate* button in order to generate the build files for your target system. These build files can then be found in the target directory you entered at the beginning (by default the `build` subdirectory of the tutorial).

Step 1 (Alternative): Using CMake from Command Line

In case you don't have a CMake GUI available you can also run CMake from command line. Since this will be mainly the case on Linux installations the following instructions are focused on this

operating system.
Before running CMake you should at first configure the paths and build options in the file `user.cmake`. For example in order to define the path of your *inVRs* installation you can uncomment the following entry:

```
# DEFINES INVRS DIRECTORY
# By uncommenting the following line you can specify the path where your INVRS
# installation is located.
# If this entry is not set cmake tries to find the path by itself.
set (inVRs_ROOT_DIR /opt/inVRs_v1.0a5/)
```

After you finished the configuration enter the path where you want your build files to be created, the recommended path is the `build` subdirectory, and call the cmake command:

```
# from GoingImmersive source directory
cd build
cmake ../
```

This should create the Makefiles in the `build` directory.

Step 2: Building the Tutorial

After the first step was finished the tutorial can be built. Therefore use your default IDE or build program. The build files can be found in the selected build target folder, which is the `build` subdirectory by default. For example when building on Linux just enter the `build` directory and call make:

```
# from GoingImmersive source directory
cd build
make
```

Starting the Application

Before the application can be started the path to the *inVRs* libraries has to be configured. This has to be done in the file `general.xml` which can be found in the subfolders `config` and `final/config`. Inside these folder you have to enter the path to your *inVRs* library directory in the following line:

```
    <path name="Plugins" directory="/please/insert/your/inVRs/libs/path/here/"/>
```

Listing 7.1: general.xml

For starting the built application two script files are available, the `startTutorial.sh` (or `startTutorial.bat` file which starts the tutorial built from the `src` directory and the `startFinalTutorial.sh` (or `startFinalTutorial.bat` which starts the final version of the tutorial which is contained in the folder `final`. Before executing these scripts you will have to open them and set the paths to the *inVRs* and *OpenSG* libraries accordingly, like:

```
# set opensg library path
export OPENSG_LIB_PATH=/usr/local/lib/opt
# set inVRs library path
export INVRS_LIB_PATH=/opt/inVRs_v1.0a5/lib
```