interactive networked virtual reality system

–

Developing VR Applications With The inVRs Framework

–

Tutorial Notes

–

IEEE Virtual Reality 2010

Christoph Anthes, Roland Landertshamer,
and Marina Lenger

March 12, 2010

Abstract

The *inVRs* framework was created to ease the design and the development of Networked Virtual Environments. This document provides a brief introduction on the concepts used by *inVRs*. It demonstrates how to develop the first application using basic navigation, interaction and network communication with a hands-on example.
After going through the tutorial, the user will be able to navigate through the virtual world, pick up objects and place them. By pressing a button an animation sequence can be started.
Typically VR applications make use of stereoscopic multi-display installations and tracking systems as well as a variety of exotic input devices. The second half of 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 scene which is if available displayed on VR hardware. Tracking systems will allow the user to display the pose of it's avatar approximately correct according to a head sensor and a sensor attached to an input device.

Contents

Chapter 1

Introduction

Networked Virtual Environments (NVEs) are getting more and more attention from the industry and research facilities. A vast amount of application areas ranging from psychology, architecture, training, scientific visualization over to art and entertainment are using Virtual Reality (VR) technology.
But it is still challenging to develop such applications since many aspects from a variety of research areas have to be taken into account. Software design, hardware development and human factors are to be considered for the creation of efficient NVEs.
Most VR applications still follow a low-level approach, where the NVE is tailored specific to the application domain.
The use of scene graphs in combination with a complete application development is still cumbersome since often the mechanisms for interaction, navigation and synchronization are reinvented. On the other hand VR applications can be created using existing NVEs, relying on scripting languages or authoring environments where graphical editors ease the design and development process. One of the major drawbacks of these solutions is their restrictiveness.
To overcome these mentioned issues and to ease the design and development process of NVEs the *inVRs* framework was developed. *inVRs* provides a structured approach using well-known software patterns. It is designed to formalize and reuse common interaction techniques and navigation methodologies, with the feature of automatic network distribution, by keeping up the needed flexibility of the low-level solutions.
Additional tools have been created for *inVRs* like the support of physics engines, a graphical editor for the layout of a VE, and a 3D widget system inside virtual worlds. Through the approach chosen for the internal communication and the network module the out-of-the-box feature of concurrent object manipulation is supported.
The *inVRs* framework is publicly available under an LGPL license at http://www.invrs.org/. It has been developed from 2005 till 2009. A large number of researchers and computer science students contributed to the code base of the framework and have created over 10 applications from the domains of new media art, architecture, entertainment, safety training, product presentation, and scientific visualization.
Currently *inVRs* supports OpenSG[1] as a scene graph, OpenAL[2] for the audio layer and ODE (**O**pen **D**ynamics **E**ngine)[3] for physics simulation. 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

[1]http://www.opensg.org/
[2]http://www.openal.org/
[3]http://www.ode.org/

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.
The tutorial consists of three consecutive parts, where the first part introduces the basic concepts of the framework. In the second part which is available independently as the *Medieval Town Tutorial* the users will learn how to create basic *inVRs* applications. With the basic knowledge of the *inVRs* framework the readers will now be able to extend their knowledge into the immersive sector in the part available independently as well as the *Going Immersive Tutorial*.
Many aspects of the *Medieval Town* part of the 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

The presented tutorial illustrates a set of key concepts of the *inVRs* framework and demonstrates easy and consistent application development. The focus is set on fast application prototyping by using the already existing modules and the system core. Writing individual components or enhancing already available components requires a deeper insight in the framework and is therefore consequentially left out in the first part of the tutorial.
Initially the reader will learn how to wrap up the cumbersome configuration setup which was explained in detail in the first part of the 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.
During the different stages of the course XML configuration data has to be altered and own C++ code has to be developed. The tutorial is organized into the following constitutive sections:

- Architecture Overview
- Basic Application Development
- Navigation and Skybox
- Transformation Management
- Interaction
- Using Network Communication
- Developing own Application Logic
- Wrapping Functionality
- Immersive Displays
- Using the Input Interface
- Coordinate Systems
- Tracking and Avatars

In the architecture chapter an overview on the provided concepts is presented and the terminology which is used in the framework is introduced. The following six chapters of the tutorial demonstrate to the participants how an interactive NVE can be configured and created using the

inVRs framework. Finally own application logic is integrated which illustrates the flexibility of the provided software.
After finishing the tutorial one should be able to navigate, with terrain following and collision detection through a shared virtual world representing a medieval town. Objects can be moved in this town and animations like starting the rotation of a windmill sail can be triggered.
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.

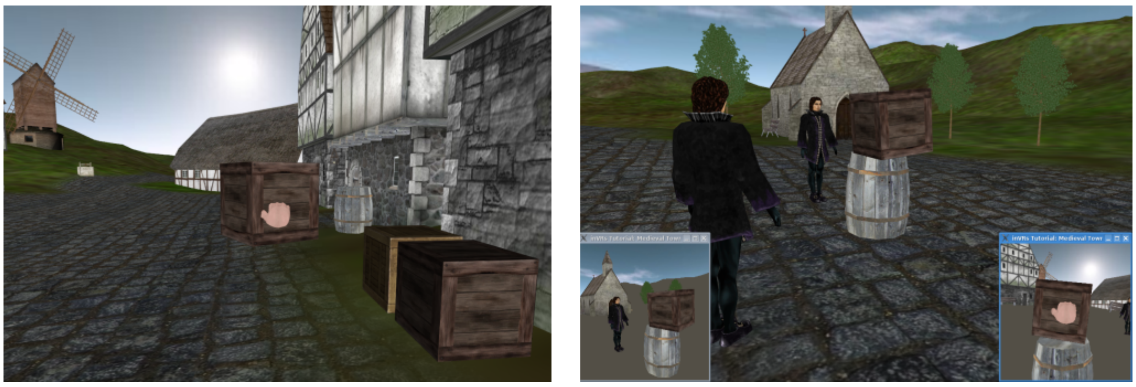


Figure 1.1: The Tutorial Town

Figure 1.1 shows the tutorial application with environments, a skybox and some houses and boxes. Collision with the surroundings is detected and interaction with these boxes is possible.

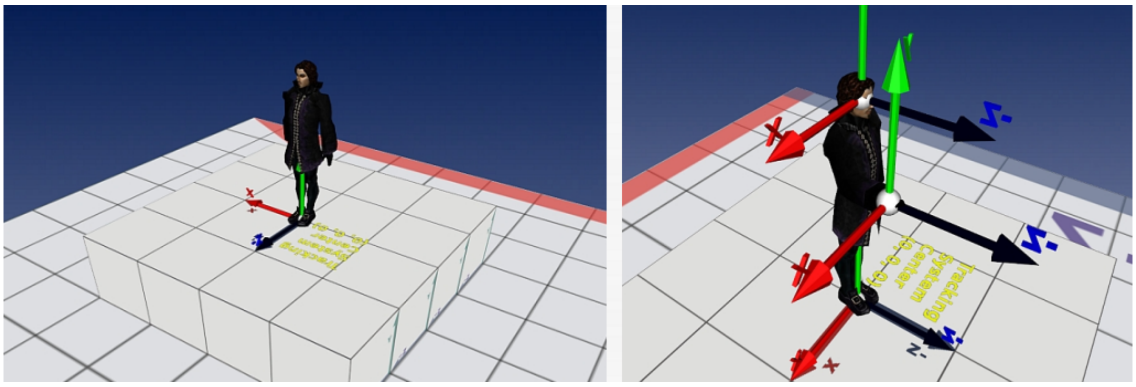


Figure 1.2: The Going Immersive Application

Figure 1.2 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 - Architecture Overview
  The architecture of the framework is presented. The *inVRs* specific terminology, the individ-

ual components like the core, the modules, and the interfaces as well as their interconnections are briefly introduced.

- Chapter 3 - Basic Application Development
  A very simple OpenSG application with an empty scene graph is described. To include *inVRs* functionality the configuration of the framework is presented and the world database is connected to the OpenSG example. Placing entities inside an environment coordinate system in the VE creates a basic scene, displaying a medieval town square. Configurations of the world database are altered in order to arrange the scene.

- Chapter 4 - Navigation and Skybox
  A flying navigation is established in order to observe the scene. The models for speed, direction and orientation of the navigation module are configured. Internally they are interconnected with the keyboard and mouse input, which is abstracted by the input interface. Basic OpenSG navigation has to be decoupled from the VE. To create the illusion of a large world additionally a skybox illustrating the surroundings of the world is set up.
  Users are now able to move the camera and their user representations, the avatars, throughout the VE. The navigation module has to be integrated in the application and interconnected to the *inVRs* system core.

- Chapter 5 - Transformation Management
  The transformation management is used in order to create terrain following and collision detection with the environment and its entities.
  The configuration of the transformation pipe is performed by integrating height map and collision map modifiers. These modifiers alter the transformations received from the navigation module. The resulting transformations are applied on the users camera and avatar.

- Chapter 6 - Interaction
  Entities in the environment can be picked and placed by using a modified HOMER (**H**and-centered **O**bject **M**anipulation **E**xtending **R**ay-casting) [BH97] interaction technique. The mouse or the touchpad is used as a conventional input device in order to select and manipulate objects in the VE.
  To create this type of interaction the transition functions of the interaction modules' state machine have to be configured and the module has to be interconnected with the system core.

- Chapter 7 - Using Network Communication
  In order to develop an NVE the network module of the framework is integrated. By connecting the network module to the core and distributing the events and transformation data the participants of the tutorial are now able to interact in a shared VE. Remote interaction can be perceived.
  To enable the distribution of transformations the transformation pipes have to be altered again.

- Chapter 8 - Developing own Application Logic
  To demonstrate the flexibility of the framework and the possibility of low-level development simple application logic is created in C++ to implement animations inside the virtual world. With simple actions and few transformation calculations the rotation of a windmill wheel can be triggered and terminated.

- Chapter 9 - 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 10 - 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 11 - 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 12 - 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 13 - 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 14 - Outlook
  The taught aspects of *inVRs* are recaptured and a brief outlook on what else could be explored using the framework is given. The additional tools as well as the available documentation are briefly introduced.

Chapter 2

Architecture Overview

inVRs consists of input and output interfaces for the interconnection to different scene graphs and the access of a variety of input devices. Three independent modules support interaction, navigation and network communication. The modules and the interfaces are connected to a system core, which manages communication between the components using discrete events and continuous flows of transformation data packets. Inside the system core a data storage system keeps the logical entities of the NVE as well as data about the users interconnected with each other.

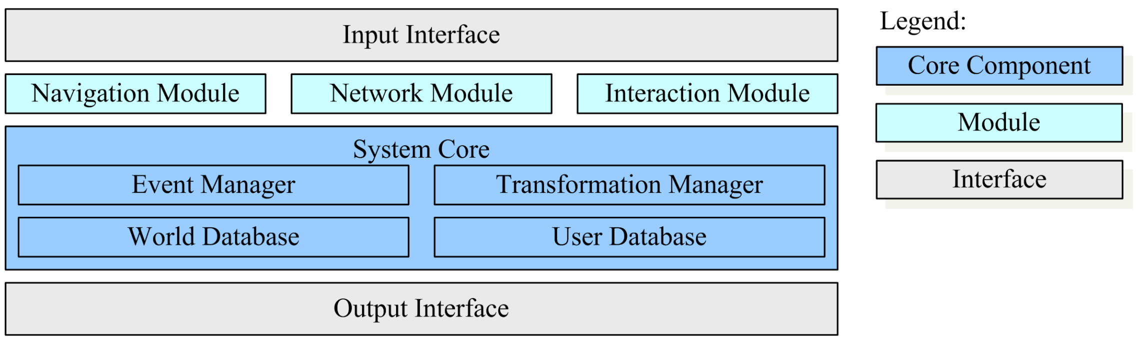


Figure 2.1: The Basic *inVRs* Components

In general three main types of components exist. Figure 2.1 illustrates the main components of the *inVRs* framework. The interfaces for input and output are shown in grey, the modules are drawn in light blue while the system core with its subcomponents is displayed in a darker blue. For future reference a short description of the components is given in Table 2.1.
Additionally many smaller features like logging functionality, math functions, and data types are integrated in the system core. More detail on the overall architecture has been previously published in [AV06].

2.1 System Core

The system core is the key library of the *inVRs* framework. It hosts the communication mechanisms in form of an event and a transformation manager and stores data of the VE in the world database and the user database.

Component	Type	Short Description
Input Interface	Interface	Handles input devices
Output Interface	Interface	Generates audio and multi-display graphics output
User Database	Core Component	Information about the local and remote users
World Database	Core Component	Information about the VE
Event Manager	Core Component	Handles discrete events
Transformation Manager	Core Component	Handles a continuous stream of transformation data packets
Core Functions	Core Component	Set of functions for extrapolation, logging, etc.
Interaction	Module	Interaction processing
Navigation	Module	Navigation trough the VE
Network	Module	Distribution of messages via the network
Tools	Add-On	Graphical effects, physics, menus, editor

Table 2.1: Components of the *inVRs* framework

2.1.1 Databases

The world database is responsible for keeping the layout of the VE including the description of its components. It acts as a high-level manager for the geometrical transformations of the VE objects. Several types of objects are available in an *inVRs* virtual world.
So-called environments do not have a graphical representation. They are coordinate systems that are used for grouping and thus the support of culling sub-objects. These environments are often used in conjunction with the network modules to split the NVE into several sub VEs.
These sub-objects could be either tiles or entities. Tiles are always fixed they can be used as decorative parts of the VE representing buildings, parts of a landscape or other static objects. Tiling mechanisms can also be used in the framework to split large datasets into disjoint parts.
The most interesting objects in the world database are the entities, which are typically used for interaction. To develop complex virtual worlds it is common to define own entity types and equip them with application specific functionality.
In Figure 2.2 the scene graph representation of environments, tiles and entities is illustrated.
The second database - the user database - manages the local and the remote users of the VE. It keeps the coordinate systems of the user representations including the cursor data and links these transformations onto the graphical representations stored inside world database of the system core.

2.1.2 Communication

The communication architecture of the *inVRs* framework differs significantly from other solutions in the field. The framework makes a clear distinction between two types of data - events and transformation data packets.
Transformation data packets contain geometrical transformations, which can be applied on objects of the VE like entities or the camera. The transformation manager handles these packets. It is not only used for the distribution of the data, but rather performs significant modification of the transformation by piping the packets through different stages of the modification process.
The events are discrete messages. Events are to be distributed in order from component to component. Event cascades where one event triggers another are to be avoided by the application designer.
In case a network module is available transformation data is typically transmitted via UDP in an unreliable manner and events are transmitted via TCP in a reliable way. The concepts of transformation management and the event system have been previously published in [ALBV07].

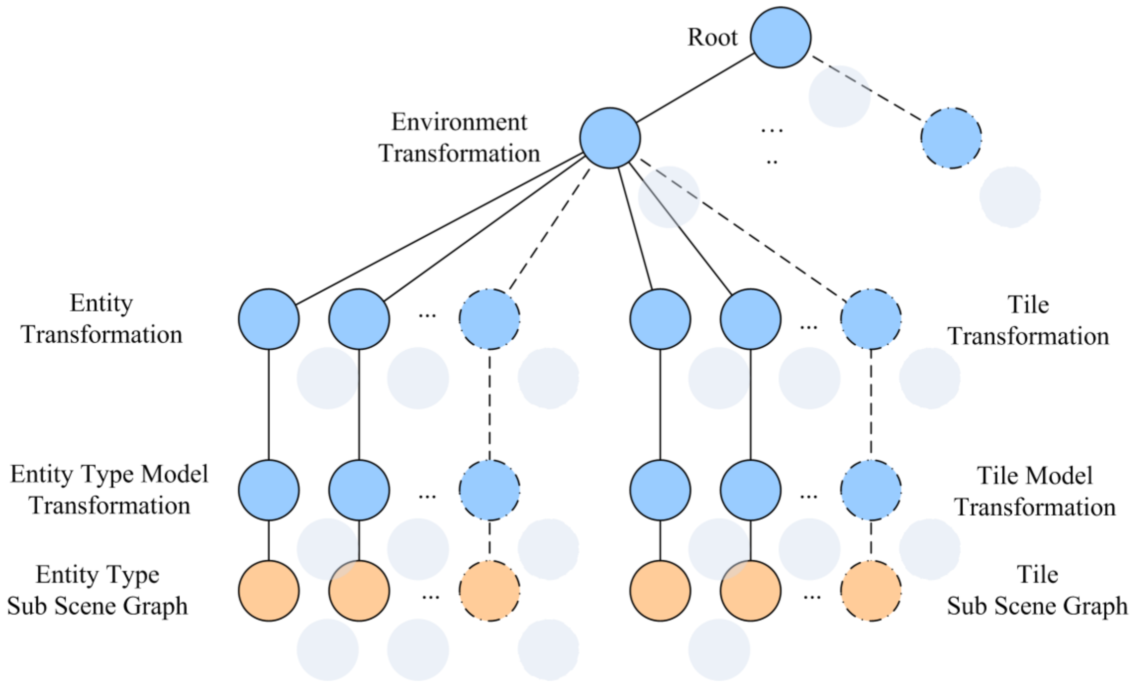

Root
Environment Transformation
Entity Transformation
Tile Transformation
Entity Type Model Transformation
Tile Model Transformation
Entity Type Sub Scene Graph
Tile Sub Scene Graph

Figure 2.2: The Transformation Hierarchy of the World Database

2.2 Interfaces

The interfaces of *inVRs* are used to abstract input and output devices. Input from tracked devices as well as standard input from mouse, keyboard and joystick can be parsed and processed by the input interface.

2.2.1 Input Interface

A mapping from the data gathered by the input devices is performed on an abstract controller, which can be accessed from the modules or the application. The input data is processed, abstracted and exposed in the form of buttons, axes and sensors.

2.2.2 Output Interface

The current implementation of *inVRs* provides an abstraction layer for scene graphs and audio output. By using the OpenSG multi-display capabilities it is easily possible to render *inVRs* applications on CAVE-like [CNSD$^+$92] devices or curved installations like the i-Cone [SG02] as well as simple monoscopic desktop systems.
For audio output currently only OpenAL is supported. The basic functionality of playing and stopping audio files is provided so far.

2.3 Modules

The modules of the framework can be loaded individually as plugins. Three basic modules implement the key features of an NVE. They handle interaction, navigation and network communication. In general an own user-defined module can replace each module as long as the common interfaces to the system core are kept. Additional modules as for example for physics simulation or animation have been successfully integrated into the framework.

2.3.1 Navigation

In the *inVRs* navigation module navigation or travel is composed by three independent models, which describe speed, orientation and direction. The models parse abstracted pre-processed data from the input interface and return a scale, a quaternion and a vector which are combined by the module to a resulting transformation matrix describing the desired offset to the last processed transformation.
This matrix is typically applied via the transformation manager either on the camera or the avatar. In the transformation pipe it is common to alter the transformation matrix received from the navigation module.

2.3.2 Interaction

In the context of the *inVRs* framework interaction is implemented as a state machine with the three states idle, selection and manipulation. In order to implement common interaction techniques transition functions have to be developed or chosen from a set of pre-defined functions.
By configuring the transition functions new interaction techniques can be developed. As an example the selection process can be exchanged from a virtual hand selection to a ray-casting selection, while the manipulation could be kept to a virtual hand manipulation.

2.3.3 Network

The network module is implemented using a two-layered approach. The top layer the high-level interface provides common access to all *inVRs* and application specific components. User defined messages, events and transformation data packets can be distributed to all other participants or a defined Area of Interest (AOI).
The low-level component of the module is designed to be exchanged and to implement specific network protocols. The communication topology and the database distribution topology is fully implemented in the low-level component and hidden from the application developer. Additional optimizations like AOI management are handled as well by the low-level component.

Chapter 3

Basic Application Development

To start with the tutorial a set of predefined code (.cpp) and configuration (.xml) files is provided. Each chapter of the tutorial contains code examples, so called snippets, which can be cut'n'pasted from the code files or this document into your main file `MedievalTown.cpp` at the places where the comments referring to these snippets are placed. The XML configuration files will have to be altered as well using the snippet mechanism.
You will find for each chapter two separate snippet files. The files for altering the code as well as the configuration changes can be found in the `snippets/` subdirectory. Under each listing provided in this document you will find the name of the source file as well as a reference to the according snippet and the destination file where it has to be pasted.
The initial file to begin with is `MedievalTown.cpp` which you should open now with the editor of your choice. An additional Eclipse project for the tutorial is available in the same directory.
You will see an OpenSG application with a set of pre-defined functions including `main`(). For convenience all needed headers as well as the global variables are already included and defined. The following list gives an overview on our pre-defined functions.

- `void cleanup()`

  The method performs a system cleanup for OpenSG and later for *inVRs* all allocated memory should be set free here.

- `void display(void)`

  This method contains the main display loop which is invoked by a GLUT callback.

- `void reshape(int w, int h)`

  The method is used for reaction on changing of the window size.

- `void mouse(int button, int state, int x, int y)`

  This method reacts to button presses of the mouse.

- `void motion(int x, int y)`

  The method forwards the coordinates of the mouse during mouse motion.

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

  This method reacts to keyboard input.

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

  The method reacts on keyboard input. It is invoked when a key is released.

- `int setupGLUT(int *argc, char *argv[])`

  This method sets up the GLUT system and registers required callback functions for example for display.

Compile the medieval town application and execute it. You should now see a simple black window. In the next steps we will explain what is happening in the `main()`-function and how *inVRs* can be used to display a VE.

3.1 Using OpenSG and GLUT

Let's have a brief look at the `main()`-function as it is. The first lines up to the `init()`-function of main are used to initialize OpenSG as well as GLUT. A GLUT window is created and a connection between the window and OpenSG is established.

```
int main(int argc, char **argv) {
  osgInit(argc, argv);                  // initialize OpenSG
  int winid = setupGLUT(&argc, argv);   // initialize GLUT

  // the connection between GLUT and OpenSG is established
  GLUTWindowPtr gwin = GLUTWindow::create();
  gwin->setId(winid);
  gwin->init();
```

Listing 3.1: MedievalTown.cpp - Top Part of `main()`

In the next part of the `main()`-function a very basic scene graph operation is performed. An OpenSG `Node` is created and filled with a `Group Core` to give it grouping functionality.
The OpenSG `SimpleSceneManager` is instantiated and the previously created window is attached to it. The newly created node is set as the root node of the scene graph. Afterwards we tell the `SimpleSceneManager` to show the whole scene. The near clipping plane of the manager is set to 0.1 since it is more convenient for our application.
The rendering of the scene can now start by invoking the `glutMainLoop()`-function.

```
  NodePtr root = Node::create();
  beginEditCP(root);
    root->setCore(Group::create());
  endEditCP(root);

  mgr = new SimpleSceneManager;  // create the SimpleSceneManager
  mgr->setWindow(gwin);          // tell the manager what to manage
  mgr->setRoot(root);            // attach the scenegraph to the  root node
  mgr->showAll();                // show the whole scene
  mgr->getCamera()->setNear(0.1);

  glutMainLoop(); // GLUT main loop
  return 0;
}
```

Listing 3.2: MedievalTown.cpp - Bottom Part of `main()`

With this piece of code a very basic OpenSG application without much content is available. The code provided in the example above is nearly identical to the example from the first OpenSG tutorial [Abe04]. Now it is time to integrate the *inVRs* functionality by starting with some simple configurations.

3.2 Configuring inVRs

Configuring the framework is on first sight very challenging due to the many files which can be added and altered, but large set of basic configurations and setups is already provided. Most of the configuration of the framework is available in XML files. It is highly recommended to keep the configuration file structure of an *inVRs* application close to the directory structure of the libraries

in order to easily find the desired configuration file.
An overview on the standard file structure is given in Figure 3.1. Where the configuration of the interfaces is shown in grey, the core components are illustrated in dark blue and the modules are drawn in light blue.
The following parts of the tutorial will configure some core components and modules of the framework. Configuring the interfaces becomes interesting if you are working with multi-display installations or rather exotic input devices.

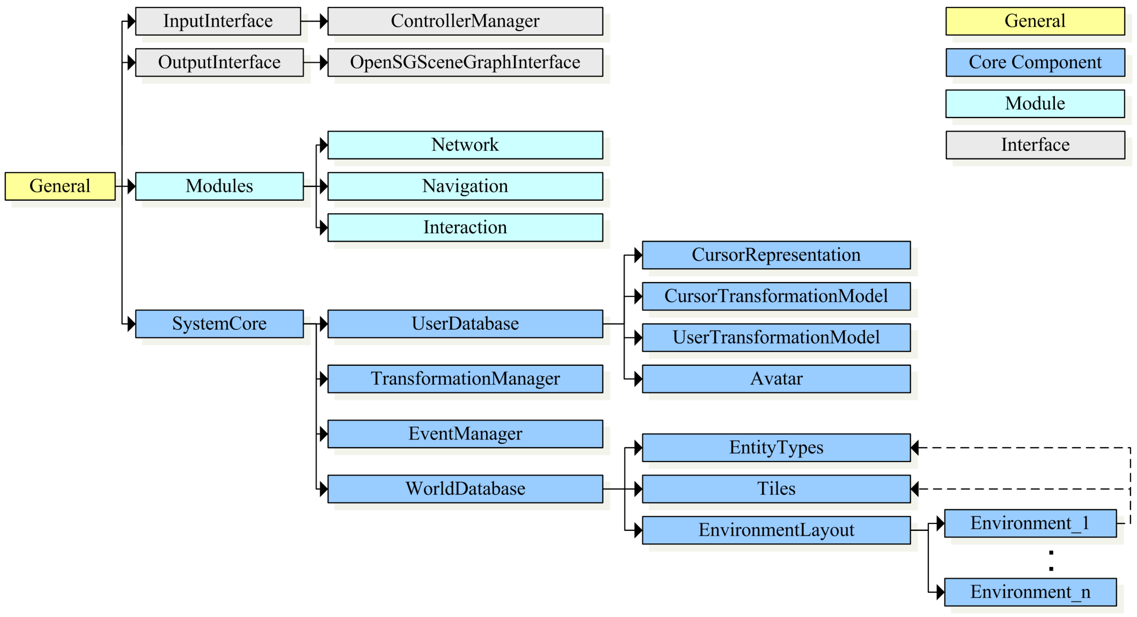


Figure 3.1: The *inVRs* Configuration Hierarchy

The first file to be looked at is the `general.xml` file, stored in the `config/` subdirectory relative to where your application lies. This file is already setup for your convenience. If we take a look at the file we will see two basic sections, the general section as well as the paths section. The general section tells us so far in which other files the actual configuration of the `SystemCore` and the `OutputInterface` is stored, while the paths refer to the storage of plugins, 3D models, textures, tool components and further *inVRs* configuration files.

```
<?xml version="1.0"?>
<!DOCTYPE generalConfig SYSTEM "http://dtd.inVRs.org/generalConfig_v1.0a4.dtd">
<generalConfig version="1.0a4">
  <!-- This is the configuration for the inVRs Framework -->
  <general>

<!-- ***************************** Snippet-2-1 ***************************** -->

    <Interfaces>
      <option key="outputInterfaceConfiguration" value="outputInterface.xml"/>
    </Interfaces>
    <SystemCore>
      <option key="systemCoreConfiguration" value="systemCore.xml"/>
    </SystemCore>
  </general>
  <paths>
    <root directory="./"/>
    <path name="Plugins"
        directory="/please/insert/your/inVRs/libs/path/here/"/>
    <path name="SystemCoreConfiguration" directory="config/systemcore/"/>
    <path name="OutputInterfaceConfiguration"
```

```
        directory="config/outputinterface/"/>

<!-- ***************************** Snippet-2-2 ***************************** -->

    <!-- Paths for World DB Datastructure-->
    <path name="WorldConfiguration"
        directory="config/systemcore/worlddatabase/"/>
    <path name="EnvironmentConfiguration"
        directory="config/systemcore/worlddatabase/environment/"/>
    <path name="EntityTypeConfiguration"
        directory="config/systemcore/worlddatabase/entity/"/>
    <path name="TileConfiguration"
        directory="config/systemcore/worlddatabase/tile/"/>

    <!-- Path for TransformationManager -->
    <path name="TransformationManagerConfiguration"
        directory="config/systemcore/transformationmanager/" />

    <!-- Paths for User DB Datastructure-->
    <path name="UserConfiguration"
        directory="config/systemcore/userdatabase/" />
    <path name="AvatarConfiguration"
        directory="config/systemcore/userdatabase/avatar/" />

<!-- ***************************** Snippet-4-6 ***************************** -->

<!-- ***************************** Snippet-2-3 ***************************** -->

<!-- ***************************** Snippet-4-1 ***************************** -->

<!-- **************************** Snippet-5-1 ****************************** -->

    <!-- Paths for Models-->
    <path name="Models" directory="models/"/>
    <path name="Tiles" directory="models/tiles/"/>
    <path name="Entities" directory="models/entities/"/>
    <path name="Skybox" directory="models/skybox/"/>
    <path name="Highlighters" directory="models/highlighters/"/>
    <path name="Avatars" directory="models/avatars/"/>
    <path name="HeightMaps" directory="models/heightmaps/"/>
    <path name="CollisionMaps" directory="models/collisionmaps/"/>
    <path name="Cursors" directory="models/cursors/"/>
  </paths>
</generalConfig>
```

Listing 3.3: general.xml

Let's leave the configuration for now and continue with the application development.
The first step in each application is to load the previously described configuration file. The static `Configuration::loadConfig()` method of the `Configuration` class is used for this purpose. This method fills the `Configuration` class with the general settings and paths from the configuration file which can lateron be accessed by the application via this class.
We should now insert the first code snippet from the file `CodeFile1.cpp` into the application, where the comment "Snippet-1-1" refers to. So please open the file `CodeFile1.cpp` stored in the `snippets/` subdirectory and cut'n'paste the parts framed by the comment into the medieval town application where you find the corresponding comment. You should proceed throughout the tutorial following the same cut'n'paste mechanism.

```
  // very first step: load the configuration of the file structures, basically
  // paths are set. The Configuration always has to be loaded first since each
  // module uses the paths set in the configuration-file
  if (!Configuration::loadConfig("config/general.xml")) {
    printf("Error: could not load config-file!\n");
    return -1;
  }
```

Listing 3.4: CodeFile1.cpp - Snippet-1-1 → MedievalTown.cpp

We do now trigger the configuration mechanism of the SystemCore by using the static method SystemCore :: configure(). This configuration mechanism will often result in subsequent configuration loading and initialization of other components. In the following code snippet the SystemCore and the OutputInterface are initialized with the configuration files read from the general section of the basic configuration file previously loaded by the Configuration class. In the configuration file for the SystemCore class references to other configuration files for the core components are located. These core components are also automatically initialized by this method call.
To combine OpenSG and *inVRs* a SceneGraphInterface has to be created. Currently only an interface to OpenSG is available but an interface to OpenSceneGraph is planned as well, just to allow for a greater flexibility. The OpenSGSceneGraphInterface is automatically loaded by the OutputInterface in the SystemCore :: configure() method.

```
  std::string systemCoreConfigFile = Configuration::getString(
      "SystemCore.systemCoreConfiguration");
  std::string outputInterfaceConfigFile = Configuration::getString(
      "Interfaces.outputInterfaceConfiguration");

  // !!!!!! Remove in tutorial part 2, Snippet-2-1 - BEGIN
  if (!SystemCore::configure(systemCoreConfigFile, outputInterfaceConfigFile)) {
    printf("Error: failed to setup SystemCore!\n");
    return -1;
  }
  // !!!!!! Remove - END
```

Listing 3.5: CodeFile1.cpp - Snippet-1-2 → MedievalTown.cpp

Inside the provided XML configuration files the layout of a medieval town was already pre-defined which has now been loaded. An OpenSG sub scene graph, in form of the top node of our scene graph, retrieved from the OpenSGSceneGraphInterface is in the next step attached to the root node of the so far empty OpenSG scene. Therefore the OpenSGSceneGraphInterface has to be obtained from the OutputInterface first. Afterwards a node is retrieved from this class which contains most part of the data defined in the configuration and stored in the WorldDatabase.
As a result we are able to change the layout of models by altering the XML file instead of changing the source. Nor do we have to recompile.

```
    OpenSGSceneGraphInterface* sgIF =
      dynamic_cast<OpenSGSceneGraphInterface*>(OutputInterface::
          getSceneGraphInterface());
    if (!sgIF) {
      printf("Error: Failed to get OpenSGSceneGraphInterface!\n");
      printf("Please check if the OutputInterface configuration is correct!\n");
      return -1;
    }
    // retrieve root node of the SceneGraphInterface (method is OpenSG specific)
    NodePtr scene = sgIF->getNodePtr();

    root->addChild(scene);
```

Listing 3.6: CodeFile1.cpp - Snippet-1-3 → MedievalTown.cpp

Finally we have to enhance the cleanup method slightly in order to remove the additional *inVRs* functionality and data structures we have created at application shutdown. Thus we have to cleanup the SystemCore.

```
SystemCore::cleanup(); // clean up SystemCore and registered components
```

Listing 3.7: CodeFile1.cpp - Snippet-1-4 → MedievalTown.cpp

Before we can run our application now we have to change the path entry ”Plugins” in the main configuration file. The *inVRs* modules and interfaces are implemented as plugins and can be dynamically exchanged. Therefore the path to the libraries in which the modules and interfaces are stored have to be set.
Please make sure to alter this path and provide the proper path describing where the libraries are stored. Usually these libraries are located in the `lib` subdirectory of the *inVRs* main directory.

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

Listing 3.8: general.xml - Enter Path to inVRs Libraries

If we recompile and start our application now we should be able to see a medieval town. This is due to the use of the `WorldDatabase` which will be explained in the following section. We are able to navigate through it by using the OpenSG `SimpleSceneManager` functionality.

3.3 Working with the WorldDatabase

The `WorldDatabase` stores the definition of the objects of the VE as well as their layout. In our case it was used for storing the sub scene graphs of the medieval town.
In general the `WorldDatabase` can distinguish between these four different types of objects:

- Environments
- Tiles
- Enities
- EntityTypes

An `Environment` acts as local coordinate system for partitioning the VE into separate regions. Environments can support the scene graphs’ culling mechanisms or they might as well be used in the area of network communication for splitting one VE up and distributing it over several servers. An `Environment` can contain tiles and entities. The definition of the environment is kept in the case of our tutorial in the configuration hierarchy under `config/systemcore/worlddatabase/environment/`. Our application uses a single environment, but in general it is of course possible to have several environments. Their arrangement is stored again in an XML-file - the `environmentLayout.xml` located in the same directory.
A `Tile` is a rectangular object in the scene. The tiles perform rather a decorative function and are used in some *inVRs* applications for simplified layouting of the VE. An example for such a scenario was the creation of a race track as implemented in the netOdrom application [AWL$^+$07]. A single tile is used in this tutorial to represent the terrain. The description for this tile can be found in the `config/systemcore/worlddatabase/tiles/` directory.
An `Entity` is an interactive object which can be moved or placed arbitrarily inside the VE. When developing own applications it is very common that an own `EntityType` with application specific functionality is created. These entities are one of the key parts to create life-like and vivid virtual worlds. Their configuration is stored in `config/systemcore/worlddatabase/entities/`.
Figure 3.2 gives a top overview on the relation of these objects. Tiles, environments and entities are displayed.
In the configuration of the `WorldDatabase` references to the configuration files of its different objects, namely `Entity`, `Tile` and `Environment` are given. It is simply a wrapper pointing at the

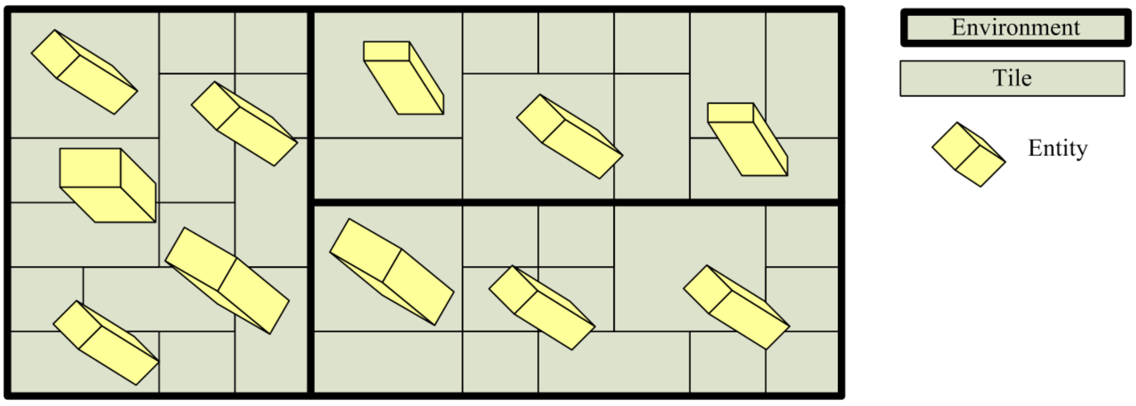


Figure 3.2: The World Database seen from Top

three major world database components. A reference to the environments is provided again in the `environmentLayout.xml` file

```
<?xml version="1.0"?>
<!DOCTYPE worldDatabase SYSTEM "http://dtd.inVRs.org/worldDatabase_v1.0a4.dtd">
<worldDatabase version="1.0a4">
  <entityTypes configFile="entities.xml"/>
  <tiles configFile="tiles.xml"/>
  <environmentLayout configFile="environmentLayout.xml"/>
</worldDatabase>
```

Listing 3.9: worldDatabase.xml

We should now take a closer look on how entities can be defined in general. No need to inspect all of our medieval town entities, but let's understand at least one of them. The configuration stored in the file `config/systemcore/worlddatabase/entity/entities.xml` describes the objects which are available in our VE. Or better it describes a certain type of entity which can be used in the VE.
In the first line the type is identified, the human-readable name of the object is given, and it is determined whether the object is considered to be fixed. The fixed attribute describes if an entity of that type is either attached to the scene or it can be selected and manipulated. In the second line the attribute representation appears, which defines, whether the sub scene graph of the model is already present with another entity type.
The next line follows with the file type and the file name of the sub scene graph which is to be loaded [1]. Providing the file type might not carry that much relevance if OpenSG is used as a scene graph, but in general it can be important.
Next an initial transformation of the representation of the entity is set. So every entity of this type is subject to this transformation, for which typically only the scale attribute is used, translation and rotation [2] are often kept to their basic values.

```
<?xml version="1.0"?>
<!DOCTYPE entityTypes SYSTEM "http://dtd.inVRs.org/entityTypes_v1.0a4.dtd">
<entityTypes version="1.0a4">
  <entityType typeId="1" name="Box01" fixed="0">
    <representation copy="false">
      <file type="VRML" name="box01.wrl"/>
      <transformation>
        <scale x="0.9" y="0.9" z="0.9"/>
```

[1]the path to theses files is stored as expected in the initial configuration
[2]which is in this case stored as a quaternion

```
      </transformation>
    </representation>
  </entityType>
...
</entityTypes>
```

Listing 3.10: entities.xml

The Tile which is used in our town is defined in `config/systemcore/worlddatabase/tile/tiles.xml`. An id as well as a human readable name is given for each tile. Two groups describe the properties of the tile and its representation in the scene.
The properties define the size in a planar dimension, while the height and the rotation can be applied to turn the tile and move it upwards. This design choice was made to provide a more intuitive layout when using an editor or manually editing the scene configuration. The representation again describes whether the tile should be copied or whether multiple references to it are allowed. Again an initial transformation as explained with the entities is applied.

```
<?xml version="1.0"?>
<!DOCTYPE tiles SYSTEM "http://dtd.inVRs.org/tiles_v1.0a4.dtd">
<tiles version="1.0a4">
  <tile id="1" name="Terrain21">
    <tileProperties>
      <size xSize="400" zSize="400"/>
      <adjustment height="0" yRotation="0"/>
    </tileProperties>
    <representation copy="false">
      <file type="OSB" name="PhysicsTestLandscape21_shader.osb"/>
      <transformation>
        <translation x="0" y="0" z="-400"/>
        <rotation x="0" y="1" z="0" angleDeg="0"/>
        <scale x="200" y="200" z="200"/>
      </transformation>
    </representation>
  </tile>
</tiles>
```

Listing 3.11: tiles.xml

The `environmentLayout.xml` file stores the arrangement of the different environments. A tile spacing is defined which is not relevant for us since we are only using a single tile in this tutorial. The same is true for the parameters xLoc and zLoc which can play in more advanced applications a role for positioning an Environment in case multiple environments are used. Typically several links to the specific environment configurations are used.
The important aspect for our tutorial application is the single reference to the file `environment.xml` which is used to store the setup as well as the arrangement of the entities which appear in our medieval town virtual world.

```
<?xml version="1.0"?>
<!DOCTYPE environmentLayout SYSTEM "http://dtd.inVRs.org/environmentLayout_v1.0a4.
    dtd">
<environmentLayout version="1.0a4">
  <tileGrid xSpacing="400" zSpacing="400"/>
  <environment id="1" configFile="environment.xml" xLoc="0" zLoc="0"/>
</environmentLayout>
```

Listing 3.12: environmentLayout.xml

If we take now a brief look at parts of `environment.xml` to which a reference was provided previously in `environmentLayout.xml`, we are able to rearrange objects.

Let's skip the details behind the map attribute, which is in the example defining a map containing a single `Tile` representing the terrain which has the id 1.
The entrypoint attributes provide an initial transformation which tells us where the user enters the VE and in which direction he looks. They are basically a classical viewpoint setting.
The entity attributes describe the transformation of an `Entity` on an per entity basis rather than on an `EnitityType` basis. In the context of object oriented languages the entity type can be compared to a class whereas an entity described in an environment represents an instance of this class.
Additionally a unique entity id has to be provided. With this id the entity can be accessed later on inside the application. In case entities are created during runtime they receive a unique id gathered from an idpool.

```
<?xml version="1.0"?>
<!DOCTYPE environment SYSTEM "http://dtd.inVRs.org/environment_v1.0a4.dtd">
<environment version="1.0a4">
  <tileMap xDimension="1" zDimension="1">
    1
  </tileMap>
  <entryPoint xPos="268" yPos="30" zPos="183" xDir="-1" yDir="0" zDir="0.4"/>
  <entity id="1" typeId="63">
    <transformation>
      <translation x="228.52" y="9.40" z="177.17"/>
      <rotation x="1.00" y="0.00" z="0.00" angleDeg="0.00"/>
      <scale x="1.00" y="1.00" z="1.00"/>
    </transformation>
  </entity>
  <entity id="2" typeId="64">
    <transformation>
      <translation x="219.24" y="9.10" z="177.49"/>
      <rotation x="1.00" y="0.00" z="0.00" angleDeg="0.00"/>
      <scale x="1.00" y="1.00" z="1.00"/>
    </transformation>
  </entity>
...
</environment>
```

Listing 3.13: environment.xml

Figure 3.3 shows the virtual world we have created so far.

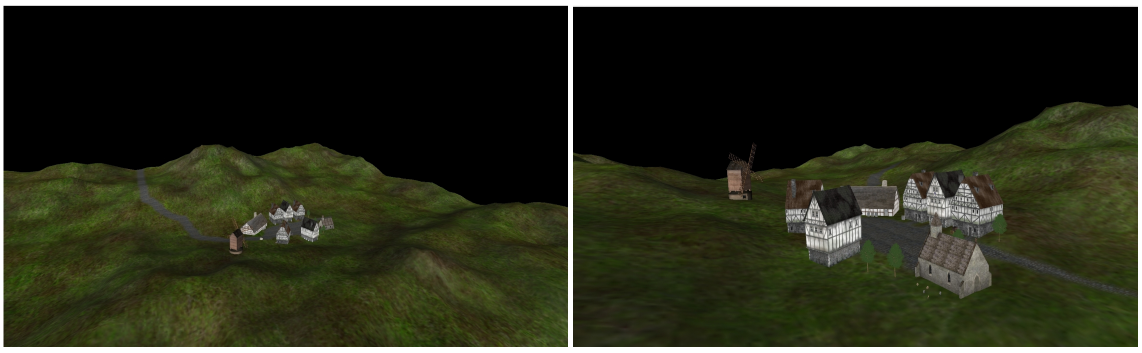


Figure 3.3: The virtual medieval town

3.4 Summary

So far we have not done that much, but we we're already able to display a scene and layout it with simple XML definitions. Initially a basic OpenSG application was created which was enhanced

with the `WorldDatabase` for simplified world and respectively scene graph layout. Loading and configuring the core was briefly demonstrated, while the description of the world database should give us now an idea on how to define a VE using *inVRs*.
Only one component of the `SystemCore` has been used yet and things will get more exciting soon when the scene becomes interactive. In the next chapter we will learn about navigation, user input and skyboxes.

Chapter 4

Navigation and Skybox

So far we are able to change the layout of our world via an XML specification, which is not too exciting yet. The navigation or travel through the scene you have experienced in the previous part is provided by the `SimpleSceneManager` of OpenSG. This navigation is now to be changed with the help of the `Navigation` module of the *inVRs* framework.
To achieve this we have to register additional components, like the input interface for parsing user input, and the navigation module, to the core. Other core components besides the world database have not been used so far, but they will in this chapter. The transformation management and the user database will be used in conjunction with the navigation module. Significant configuration has to take place in order to include the components.
Additionally we thought it is nice to have a brighter environment with sky and clouds surrounding our medieval town. Such a visual improvement is typically achieved in the area of games or similar virtual worlds by the use of skyboxes which are going to be introduced as well in this chapter.

4.1 Adding inVRs Components

Let's get started with the implementation and configuration of the navigation by adding the needed components. In order to use *inVRs* components in general they have to be registered at the core. But besides registration at the core the components might need to trigger user defined functionality during their initialization. Thus initialization callbacks have to be registered initially as well.
The additional components we are going to use now are the `Controller` of the InputInterface for gathering user input and the `Navigation` module for the processing of this input. In order to pass the calculated `TransformationData` to the camera we use the `TransformationManager` of the core, which has to be configured. We do need another core component the `UserDatabase` which contains user transformations and the camera transformations. This database is already fully set up and will furthermore not be described in this tutorial.
An overloaded configure method is to be called in order to pass all loaded configuration files, in our case one for each *inVRs* component type. This configure method triggers the whole system configuration and will load the plugins and invoke registered callbacks as we will see later.
Watch out if you insert the following snippet in the code. The original configure method as used in the previous section has to be removed as indicated in the comment.

```
// !!!!!! Remove part of Snippet-1-2 (right above)
// in addition to the SystemCore config file, modules and interfaces config
// files have to be loaded.
std::string modulesConfigFile = Configuration::getString(
      "Modules.modulesConfiguration");
std::string inputInterfaceConfigFile = Configuration::getString(
    "Interfaces.inputInterfaceConfiguration");
```

```
  if (!SystemCore::configure(systemCoreConfigFile, outputInterfaceConfigFile,
      inputInterfaceConfigFile, modulesConfigFile)) {
    printf("Error: failed to setup SystemCore!\n");
    printf("Please check if the Plugins-path is correctly set to the inVRs-lib
        directory in the ");
    printf("'final/config/general.xml' config file, e.g.:\n");
    printf("<path name=\"Plugins\" path=\"/home/guest/inVRs/lib\"/>\n");
    return -1;
  }
```

Listing 4.1: CodeFile2.cpp - Snippet-2-1 → MedievalTown.cpp

We have to update as well the general configuration stored in `general.xml` and provide settings defining in which files the configurations of the newly integrated component types are located.

```
    <Modules>
      <option key="modulesConfiguration" value="modules.xml" />
    </Modules>
    <Interfaces>
      <option key="inputInterfaceConfiguration" value="inputInterface.xml" />
    </Interfaces>
```

Listing 4.2: Snippets2.xml - Snippet-2-1 → general.xml

Additionally the paths for the configurations of the modules and the `InputInterface` have to be defined.

```
    <path name="InputInterfaceConfiguration" directory="config/inputinterface/" />
    <path name="ModulesConfiguration" directory="config/modules/" />
```

Listing 4.3: Snippets2.xml - Snippet-2-2 → general.xml

In order to parse the configurations of the individual components the paths to their configuration files have to be set as well in the `general.xml` configuration file.

```
    <!-- Path for Interfaces Datastructure -->
    <path name="ControllerManagerConfiguration"
      directory="config/inputinterface/controllermanager/" />

    <!-- Paths for Module Datastructure -->
    <path name="NavigationModuleConfiguration"
        directory="config/modules/navigation/" />
```

Listing 4.4: Snippets2.xml - Snippet-2-3 → general.xml

In order to have access to the different interfaces and components during the initialization and configuration phase callback functions have to be defined. They will be triggered at the initialization of the `SystemCore`. The `initInputInterface()`-method is registered as a callback function. During the initialization it is important to get a pointer to the interface, or more specific in our case a pointer to the controller, in order to access it later on in the application.

```
void initInputInterface(ModuleInterface* moduleInterface) {
  // store ControllerManger and the Controller as soon as the ControllerManager
  // is initialized
  if (moduleInterface->getName() == "ControllerManager") {
    controllerManager = (ControllerManager*)moduleInterface;
    controller = (Controller*)controllerManager->getController();
  }
}
```

Listing 4.5: CodeFile2.cpp - Snippet-2-2 → MedievalTown.cpp

The `initModules()`-method is invoked as well by a callback. The design approach is analog to the previous introduced method. It takes care of the modules instead of the interfaces. As you can see this snippet includes references to other snippets. In the placeholder other modules will have to be registered in later chapters.

```
void initModules(ModuleInterface* module) {
  // store the Navigation as soon as it is initialized
  if (module->getName() == "Navigation") {
    navigation = (Navigation*)module;
  }
  //----------------------------------------------------------------------//
  // Snippet-4-1                                                          //
  //----------------------------------------------------------------------//

  //----------------------------------------------------------------------//
  // Snippet-5-1                                                          //
  //----------------------------------------------------------------------//
}
```

Listing 4.6: CodeFile2.cpp - Snippet-2-3 → MedievalTown.cpp

Now we take care of the needed component types and register our previously defined callback functions at the *inVRs* `SystemCore` and `InputInterface`.

```
  // register callbacks
  InputInterface::registerModuleInitCallback(initInputInterface);
  SystemCore::registerModuleInitCallback(initModules);
```

Listing 4.7: CodeFile2.cpp - Snippet-2-4 → MedievalTown.cpp

This whole setup of the configuration files might seem to be unintuitive and a lot of work at the beginning, but it is needed to provide full flexibility. Once one has established the configuration structure most developed setups can be used later on for future applications.
Having now finally finished our component setup it is time to start with the actual navigation implementation.

4.2 Navigation

The approach *inVRs* uses for navigation might seem fairly unconventional compared to straight forward hard coded implementations of navigation techniques, but it comes with many advantages especially in the areas or reusability and structure if we look at other solutions. Navigation in the context of the *inVRs* framework is composed by three independent parts: speed, orientation, and direction. These different aspects are implemented as individual models (`SpeedModel`, `OrientationModel`, `DirectionModel`) that are combined in order to generate a resulting matrix. This transformation matrix which is stored in form of a `TransformationData` packet contains information about the new position and orientation of the object which is being bound to the navigation.
The object which is controlled by the navigation does not necessarily have to be the camera. It could be for example as well an avatar, to which a dangling camera is attached like it is often the case in third person computer games.
More details on the navigation composition approach and its individual models can be found in [AHKV04].
In order to generate their results these three types of models have to gather user input. Thus they poll data from an abstract controller which is implemented inside the input interface. Describing

the Controller in a detailed way would go to far inside this part of the tutorial. Let's for simplicity just accept, that the input generated from devices is exposed in the Controller class in abstract form of buttons (providing boolean values), axes (offering 2D data) and sensors (storing 3D transformations) to all *inVRs* components and the own developed application. Callbacks can be registered as well on button presses and releases.
Before we can add the changes in the source code we still have to update some configurations. At first we have to add an avatar in our configuration which will be used to represent the user in the virtual world. A more detailed description of the avatar configuration is presented in section 7.4

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

Listing 4.8: Snippets2.xml - Snippet2-4 → userDatabase.xml

Next we have to add the Navigation module in the module configuration file so that it is automatically loaded at application startup.

```
<module name="Navigation" configFile="navigation.xml" />
```

Listing 4.9: Snippets2.xml - Snippet2-5 → modules.xml

Now that the configurations are updated we can have a look at the source code. As a first step we have to retrieve the local users avatar and camera. We have worked so far with the WorldDatabase and now it is time to access the UserDatabase. Thus the localUser is requested from the UserDatabase. Pointers to the camera and the avatar of the user are stored for later access. Since we are working in a single user environment so far the avatar is not to be displayed yet. The initial transformation for a local user is requested from the world database [1] and set on the local user.

```
// fetch users camera, it is used to tell the Navigator where we are
localUser = UserDatabase::getLocalUser();
if (!localUser) {
  printd(ERROR, "Error: Could not find localUser!\n");
  return -1;
}

camera = localUser->getCamera();
if (!camera) {
  printd(ERROR, "Error: Could not find camera!\n");
  return -1;
}

avatar = localUser->getAvatar();
if (!avatar) {
  printd(ERROR, "Error: Could not find avatar!\n");
  return -1;
}
avatar->showAvatar(false);

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

Listing 4.10: CodeFile2.cpp - Snippet-2-5 → MedievalTown.cpp

Now it is time to disconnect the OpenSGs' navigation from our VE and replace it with the navigation mechanisms of the *inVRs* framework. First we have to disable the Navigator of the

[1] you might remember we have encountered the initial transformation in the previous chapter in the environment configuration

SimpleSceneManager.
A timer is initialized which is needed later to pass the time difference between the last navigation step to the current navigation step. A common mistake is to make the navigation speed dependent on the framerate.

```
// Navigator is part of SimpleSceneManager and not of the inVRs framework
Navigator *nav = mgr->getNavigator();
nav->setMode(Navigator::NONE);      // turn off the navigator
lastTimeStamp = timer.getTime();    // initialize timestamp;
camMatrix = gmtl::MAT_IDENTITY44F; // initial setting of the camera matrix
```

Listing 4.11: CodeFile2.cpp - Snippet-2-6 → MedievalTown.cpp

The updating of the navigation module is performed once per frame. Thus the display method is modified for this update. The Controller of the input interface is internally updated by calling its update()-method. The devices [2] are polled and the new values are set. Next the Navigation has to be updated using the newly gathered values from the controller. Since the navigation transformations are passed through the TransformationManager the processing of the manager has to be invoked as well. This invocation is triggered by calling the TransformationManager::step() method, with the timer delta value describing the passed time since the previous call and a priority. The priority is needed for executing the transformations of the navigation prior to all other transformations. More detail is provided in the Programmers' Guide and the Doxygen[3] API documentation.

```
float currentTimeStamp;
Matrix osgCamMatrix;
float dt; // time difference between currentTimestamp and lastTimestamp

currentTimeStamp = timer.getTime(); //get current time
dt = currentTimeStamp - lastTimeStamp;

controller->update();    // poll/update associated devices
navigation->update(dt);  // update navigation

// process transformations which belong to the pipes with priority 0x0E000000
TransformationManager::step(dt, 0x0E000000);

camera->getCameraTransformation(camMatrix); // get camera transformation
```

Listing 4.12: CodeFile2.cpp - Snippet-2-7 → MedievalTown.cpp

The camera has to be updated where we do need a conversion from GMTL into OpenSG. The transformation of the camera has to be passed in the Navigator of the SimpleSceneManager. The step of the TransformationManager has to be invoked and one step of iterations is considered to be passed, thus the old out-of-date timestamp is replaced with the current time.

```
set(osgCamMatrix, camMatrix);       // convert gmtl matrix into OpenSG matrix

Navigator* nav = mgr->getNavigator();
nav->set(osgCamMatrix);             // plug new camera matrix into navigator

TransformationManager::step(dt);    // process the remaining pipes

lastTimeStamp = currentTimeStamp;
```

Listing 4.13: CodeFile2.cpp - Snippet-2-8 → MedievalTown.cpp

[2]in our case GlutCharKeyboardDevice and GlutMouseDevice which we will see soon
[3]http://www.invrs.org/doxygen/

4.2.1 Managing User Input

To finally access the keyboard or mouse we have to forward the input gathered by GLUT to our input interface. The `GlutMouseDevice` implements a wrapper for a GLUT mouse and can provide input to our `Controller` object which again is accessed internally by the models of the `Navigation`. Thus a fair bit of minor changes in the already provided GLUT callback functions have to be applied.
In the reshape() - function we additionally pass the current window size to the `GlutMouseDevice` which is registered at our `Controller`.

```
// the mouse device must be aware of the window size in pixel
GlutMouseDevice::setWindowSize(w, h);
```

Listing 4.14: CodeFile2.cpp - Snippet-2-9 → MedievalTown.cpp

The mouse state and the buttons have to be passed to the our newly used input processing unit, the `GlutMouseDevice` in the callback mouse(). Watch out in this example the lines above the snipped have to be replaced.

```
// !!!!!! Remove part above
// instead of calling the SimpleSceneManager we delegate the message to
// our mouse device
GlutMouseDevice::cbGlutMouse(button, state, x, y);
```

Listing 4.15: CodeFile2.cpp - Snippet-2-10 → MedievalTown.cpp

The coordinates of the mouse cursor have to be passed during movement of the mouse to the `GlutMouseDevice`. This has to take place in the motion() callback function. A similar replacement as in the previous change has to take place.

```
// !!!!!! Remove part above
// instead of calling the SimpleSceneManager we delegate the message to
// our mouse device
GlutMouseDevice::cbGlutMouseMove(x, y);
```

Listing 4.16: CodeFile2.cpp - Snippet-2-11 → MedievalTown.cpp

In the `keyboard()`-function an additional line of code has to be integrated in order to pass the key pressing to the input interface.

```
// notify keyboard device about GLUT message
GlutCharKeyboardDevice::cbGlutKeyboard(k, x, y);
```

Listing 4.17: CodeFile2.cpp - Snippet-2-12 → MedievalTown.cpp

If we move around with the mouse to change the orientation of the camera and we are in the VE we don't want to see the mouse cursor moving. It is basically attached to the window but not displayed. On the other hand we might want to move the cursor out of our current window back on the desktop. To allow switching between the modes we have to toggle the mouse grabbing. By pressing "m" or "SHIFT-m" grabbing can be toggled.

```
// grab the mouse
case 'm':
case 'M': {
  grabMouse = !grabMouse;
  GlutMouseDevice::setMouseGrabbing(grabMouse);
} break;
```

Listing 4.18: CodeFile2.cpp - Snippet-2-13 → MedievalTown.cpp

In the `keyboardUp()`-function an additional line of code has to be integrated in order to pass the key release to the `GlutCharKeyboardDevice`. This is important if we stop moving for example.

```
  GlutCharKeyboardDevice::cbGlutKeyboardUp(k, x, y);
```

Listing 4.19: CodeFile2.cpp - Snippet-2-14 → MedievalTown.cpp

In order to properly set up your navigation you have to configure the individual models for direction, speed and orientation. For configuring such a model three different aspects have to be considered, the type of the model which is used including its specific parameters. Additionally a mapping from the controller object to the navigation models has to be established in the argument attribute.
The configured mapping basically checks for the orientation the mouse movement, for the acceleration and for changing the translation the keys 'w', 'a', 's', and 'd' are looked up. This is now your new input for using the navigation technique composed by the three navigation models.

```xml
<?xml version="1.0"?>
<!DOCTYPE navigation SYSTEM "http://dtd.inVRs.org/navigation_v1.0a4.dtd">
<navigation version="1.0a4">
  <translationModel type="TranslationViewDirectionButtonStrafeModel">
    <arguments>
      <arg key="frontIndex" type="uint" value="3"/>
      <arg key="backIndex" type="uint" value="4"/>
      <arg key="leftIndex" type="uint" value="5"/>
      <arg key="rightIndex" type="uint" value="6"/>
    </arguments>
  </translationModel>
  <orientationModel type="OrientationDualAxisModel" angle="20">
    <arguments>
      <arg key="xAxisIndex" type="int" value="0"/>
      <arg key="yAxisIndex" type="int" value="1"/>
      <arg key="buttonIndex" type="int" value="1"/>
    </arguments>
  </orientationModel>
  <speedModel type="SpeedMultiButtonModel" speed="10">
    <arguments>
      <arg key="accelButtonIndices" type="string" value="3 4 5 6"/>
    </arguments>
  </speedModel>
</navigation>
```

Listing 4.20: navigation.xml

Of course this is just an example configuration for the `Navigation`. Many different models are available in *inVRs* and newly developed ones can be easily integrated. A whole set of model combinations is defined which allows easy switching between navigation techniques without altering the application code.

4.3 Skybox

Now it is time to lighten up the medieval town of our application and take it out of the dark ages by putting a nice blue sky around it.
Approaches to implement this functionality would be sky domes, just a simple blue background color or skyboxes, which is the way we follow. Skyboxes in general typically consist of six textures

that are mapped on a cube. They are used to visualize scene surroundings and the landscape at far distances. An example for such a skybox is given in Figure 4.1.

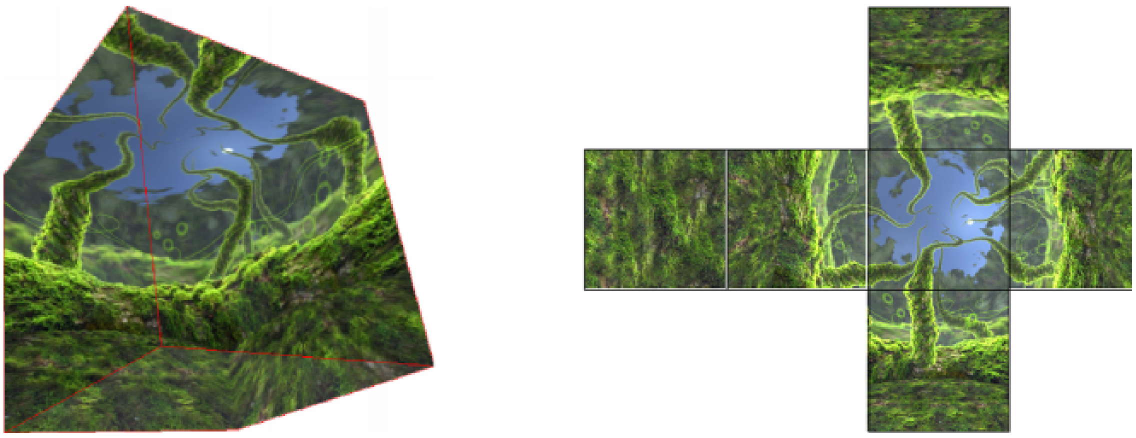


Figure 4.1: An Example Skybox

Since it is often case that the bottom or the top part of a skybox can be displayed at lower resolutions, the *inVRs* `Skybox` is allowed to have the shape of an actual box rather than a cube. The proportional dimensions of the box are set as the first three parameters of the `init()`-method of the skybox. The fourth parameter describes the distance from the camera to the far clipping plane.
In order to use a skybox, which is provided as a scene graph specific tool, the following snippet has to be included. We use again a path from the initial configuration file to find out where the textures of the skybox are located on disk.

```
// generate and configure the SkyBox
std::string skyPath = Configuration::getPath("Skybox");
skybox.init(5,5,5, 1000, (skyPath+"lostatseaday/lostatseaday_dn.jpg").c_str(),
  (skyPath+"lostatseaday/lostatseaday_up.jpg").c_str(),
  (skyPath+"lostatseaday/lostatseaday_ft.jpg").c_str(),
  (skyPath+"lostatseaday/lostatseaday_bk.jpg").c_str(),
  (skyPath+"lostatseaday/lostatseaday_rt.jpg").c_str(),
  (skyPath+"lostatseaday/lostatseaday_lf.jpg").c_str());
```

Listing 4.21: CodeFile2.cpp - Snippet-2-15 → MedievalTown.cpp

After having generated the skybox we have to attach it to the scene graph. We retrieve the OpenSG `NodePtr` of our `Skybox` object, which was internally generated by the skybox tool and attach it as a child node to the root node of the scene.

```
  // add the SkyBox to the scene
  root->addChild(skybox.getNodePtr());
```

Listing 4.22: CodeFile2.cpp - Snippet-2-16 → MedievalTown.cpp

Since skyboxes are always at the same position as the camera but they of course are attached to the orientation of the scene an immediate connection has to be established in the position attribute. Therefore the camera position is passed directly to the `Skybox` object.

```
skybox.setupRender(camera->getPosition());
```

Listing 4.23: CodeFile2.cpp - Snippet-2-17 → MedievalTown.cpp

Now it is time to recompile and execute the application again. You should now be able to see a nice blue sky around your medieval village which can also be seen in Figure 4.2. Doesn't that make you happy? Well, there is still much more to come.

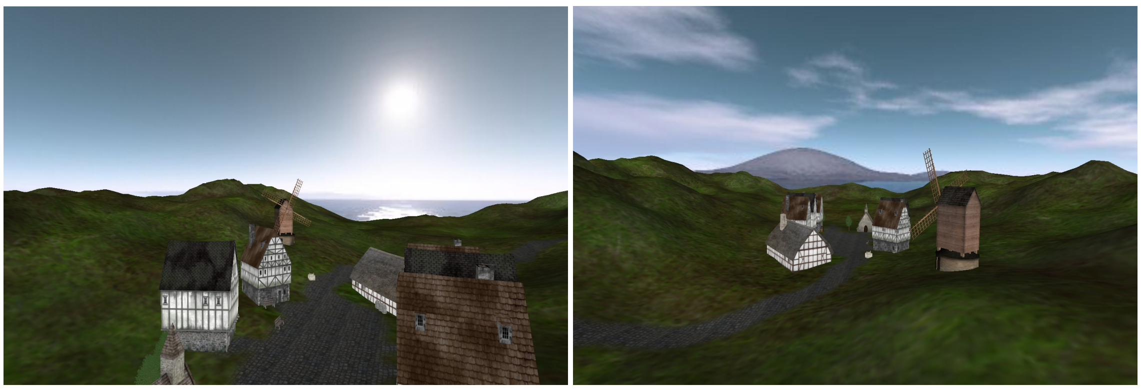


Figure 4.2: The medieval town under a blue sky

4.4 Summary

After having completed this chapter we can now use our own navigation models to control camera or basically user movement throughout the VE.
A detailed explanation on how inVRs components can be integrated has been given in order to register and configure the required modules and interfaces.
The access to the input devices has been briefly introduced by decoupling calls in the GLUT callback functions to OpenSGs' `SimpleSceneManager` and replacing it with *inVRs* devices and an abstract controller.
As you can see we are moving a fair bit away from standard OpenSG. Now we are able fly through our medieval town, by using previously defined navigation models. The town has been polished up a bit through the inclusion of an additional tool. The next step introduces gravity and collision with the scene.

Chapter 5

Transformation Management

Since we want to implement terrain following and collision detection it is useful, to work with the external tools for height and collision maps in combination with the transformation management. The transformation management is a specific concept implemented in *inVRs*. One of the two communication units of the `SystemCore` is the `TransformationManager` which is able to receive `TransformationData` packets from arbitrary *inVRs* components as well as user defined components or an application. It can modify them and afterwards distribute the packets to other components.
If we want to integrate for example gravity or collision detection in an *inVRs* VE it is recommended to use the `TransformationManager` to post-process the data received from the `Navigation` and apply it on the camera.
The key idea is to pipe transformations through the manager after having set up a pipe configuration beforehand. But before we start to explain the concepts of the modifiers, the pipes and the manager in detail, it is important to understand how the collision maps and the height maps work.

5.1 Height and Collision Maps

One of the additional features of the *inVRs* framework is a 2D Physics module which allows for fast collision detection and response [BLAV06]. Besides the module additional tools for the use of height maps and collision maps were developed.
Figure 5.1 illustrates such a height map, showing a grid containing the height values and additionally the normal vectors at the given grid positions. Height maps are ideal to implement fast terrain following. Initially these height maps have to be generated based on a an input model. They can be pre-generated offline or created dynamically during runtime.

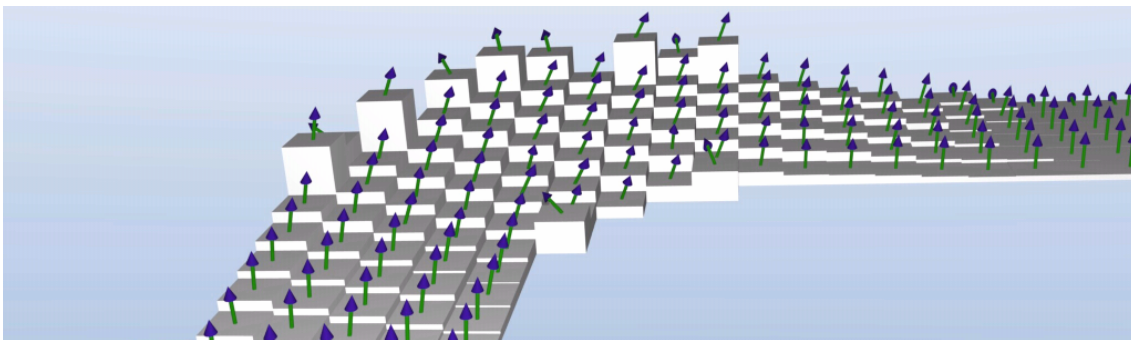


Figure 5.1: An Example Height Map

The other tool which is used are the collision maps. Collision maps are basically line sets that are mapped on a plane. Figure 5.2 shows a mesh with a generated collision map. There are many approaches to generate these maps. Tools for automatic creation have been developed but it is as well possible to model the maps manually using tools like Blender or MAYA.

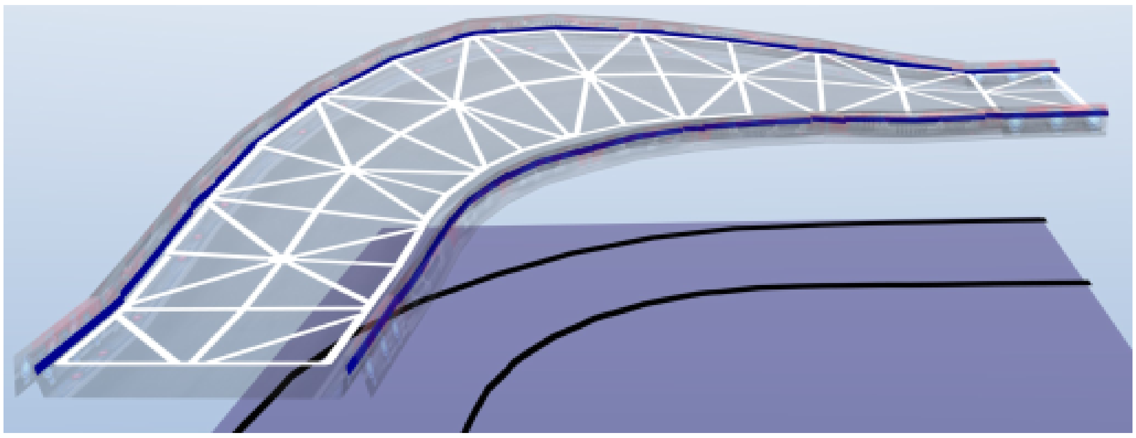


Figure 5.2: A Collision Map

5.1.1 Generating Collision Maps

The creation of collision maps is always performed offline. Two basic approaches exist for creating the desired models. The first approach is to define the collision maps manually in a 3D modelling tool like Blender. The Generation of the collision maps using this approach can be summarized in the following steps:

- Step 1:
  dump scene: start inVRs application and dump scene into file (e.g. in VRML file)
- Step 2:
  open scene in 3D modeling tool (e.g. Blender[1])
- Step 3:
  hide unneeded objects for better overview (e.g. terrain, since there is no collision with it needed)
- Step 4:
  draw linesets in top view around objects where the collisions should occure (e.g. outer walls of buildings)
- Step 5:
  export line sets into VRML97 file
- Step 6:
  enter the URL to the generated VRML file in the TransformationManager-configuration as parameter for the CollisionMapModifier

Figure 5.3 illustrates the collision map generation in Blender.
Another approach is to generate the collision maps automatically. An algorithm for this approach is described in [BLAV06].

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

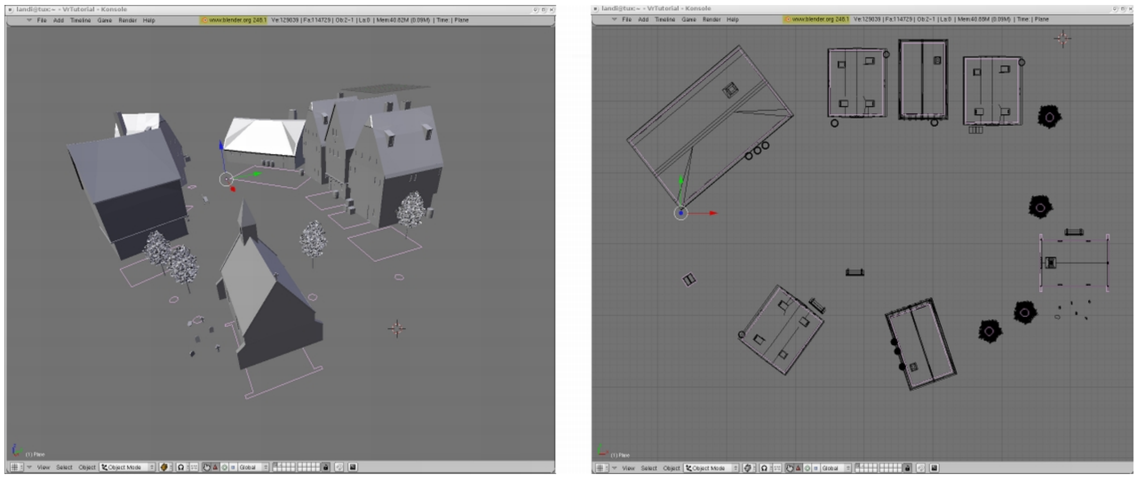


Figure 5.3: Generation of a Collision Map

5.1.2 Generating Height Maps

While the collision maps are always generated offline the height maps can follow a different approach. This is the point where we are again back in the code. The HeightMapManager has to be triggered in order to provide height maps to the system by invoking the method HeightMapManager :: generateTileHeightMaps(). The generation of these maps follows of two approaches.
First it is possible that such a height map has been previously generated, which is very convenient since it takes less time loading it than generating it on the fly.
If such a height map is not available for a tile a dynamic generation of the map will be invoked. The geometry of the pipe will be sampled and the results are stored in the height map.

```
HeightMapManager::generateTileHeightMaps();
```

Listing 5.1: CodeFile3.cpp - Snippet-3-1 → MedievalTown.cpp

5.2 Using Modifiers and Pipes

Now let's get back to the transformation management. The TransformationManager consists of several Pipes in which a variety of steps can be performed. Each pipe has a pre-defined amount of stages, these stages are called modifiers and are implemented in the TransformationModifier class. If an object (e.g. camera, entity) should be transformed via the manager a pipe between the manager and the component responsible for the object is opened. The transformations are sent by the source component to the transformation manager, where the pipe has been setup based on a key. The key is composed by several attributes like source or object type.
If we want to integrate the modifiers into the application we have to configure them in the `modifiers.xml` file which is used to set up the TransformationManager. In general we can use one pipe per object. If the object is to be transformed the pipe on this object is opened.
The configuration of a transformation pipe including the setup of the modifiers is described in an XML file. The modifier descriptions can become fairly complex so we are working now with a really simple example.
The order of the modifiers is important because they will be executed in the order they are registered. The pipes are executed in the order they are defined as well.
The pipe definition contains information about, from where and to where the data is to be sent.

The srcComponentName and dstComponentName attributes of the Pipe describe the source and target components of the pipe. If the fromNetwork attribute is set to 1 the source component has remote location, meaning the transformations were received and are entered by the local Network module. Let's not worry about the other attributes; they are relevant for more advanced applications.

```xml
<?xml version="1.0"?>
<!DOCTYPE transformationManager SYSTEM "http://dtd.inVRs.org/
    transformationManager_v1.0a4.dtd">
<transformationManager version="1.0a4">
  <mergerList/>
  <pipeList>
    <pipe srcComponentName="NavigationModule"
        dstComponentName="TransformationManager" pipeType="Any"
        objectClass="Any" objectType="Any" objectId="Any"
        fromNetwork="0">
      <modifier type="ApplyNavigationModifier"/>

<!-- **************************** Snippet-3-1 ****************************** -->

<!-- **************************** Snippet-3-2 ****************************** -->

<!-- **************************** Snippet-5-3 ****************************** -->

      <modifier type="UserTransformationWriter"/>
      <modifier type="CameraTransformationWriter">

<!-- **************************** Snippet-3-3 ****************************** -->

      </modifier>
      <modifier type="AvatarTransformationWriter">
        <arguments>
          <arg key="clipRotationToYAxis" type="bool" value="true"/>
        </arguments>
      </modifier>

<!-- **************************** Snippet-4-4 ****************************** -->

    </pipe>

<!-- **************************** Snippet-4-5 ****************************** -->

<!-- **************************** Snippet-5-4 ****************************** -->

<!-- **************************** Snippet-5-5 ****************************** -->

  </pipeList>
</transformationManager>
```

Listing 5.2: modifiers.xml

So far only these modifiers were used:

- `ApplyNavigationModifier`
- `UserTransformationWriter`
- `CameraTransformationWriter`
- `AvatarTransformationWriter`

In our case we have used a single pipe for the navigation with a very basic set of modifiers. The `ApplyNavigationModifier` inserts the results from the `Navigation` into the pipe. Afterwards the user transformation and the camera transformation are written via the two writers, the

`UserTransformationWriter` and the `CameraTransformationWriter` back to the `User`.
As a first step we have to add the `HeightMapModifier` into the pipe immediately after the `ApplyNavigationModifier` inserts its transformation into the pipe. On the transformation which has been received from the navigation module now an additional offset is applied. The height component of the transformation is set to the height value of the height map. Afterwards user and camera transformation are set

```
      <modifier type="HeightMapModifier" />
```

Listing 5.3: Snippets3.xml - Snippet-3-1 → modifiers.xml

Each type of modifier can have its own specific configuration. To use the collision maps we include the `CheckCollisionModifier`. Initially the radius based on the input transformation is checked. If the distance between a line of the collision map and the input transformation is below the defined radius the transformation form the previous pipe run is passed on in the pipe. To actually store and load the line sets the VRML file format is used. The second parameter in our example defines the file name of the file containing the collision map geometry.

```
      <modifier type="CheckCollisionModifier">
        <arguments>
          <arg key="radius" type="float" value="1" />
          <arg key="fileName" type="string" value="MedievalTownCollisionMap.wrl"/>
        </arguments>
      </modifier>
```

Listing 5.4: Snippets3.xml - Snippet-3-2 → modifiers.xml

We do not want to be hovering directly on the terrain. In order to move the camera from the actual height value of the height map we do have to apply an offset by passing additional parameters to the `CameraTransformationWriter`.

```
        <arguments>
          <arg key="cameraHeight" type="float" value="1.8"/>
          <arg key="useGlobalYAxis" type="bool" value="true"/>
        </arguments>
```

Listing 5.5: Snippets3.xml - Snippet-3-3 → modifiers.xml

After having set up our pipe and modifiers we should now come back to our C++ application. This method registers the factories for the `HeightMapModifiers` as well as the ones for the `CheckCollisionModifiers` at the `TransformationManager`. This is highly important since these modifiers are not directly stored in the `SystemCore` but rather as external tools.
Although the `TransformationManager` was previously used no callback was needed since it did not use any external components. This has changed now.

```cpp
void initCoreComponents(CoreComponents comp) {
  // register factory for HeightMapModifier as soon as the
  // TransformationManager is initialized
  if (comp == TRANSFORMATIONMANAGER) {
    TransformationManager::registerModifierFactory
      (new HeightMapModifierFactory());
  // register factory for CheckCollisionModifier
    TransformationManager::registerModifierFactory
      (new CheckCollisionModifierFactory);
  }
}
```

Listing 5.6: CodeFile3.cpp - Snippet-3-2 → MedievalTown.cpp

With this code snippet we register a callback on the initialization on the core components.

```
SystemCore::registerCoreComponentInitCallback(initCoreComponents);
```

Listing 5.7: CodeFile3.cpp - Snippet-3-3 → MedievalTown.cpp

If we recompile and execute our application, assuming we have done everything right, we should be able to move trough the medieval town. We can glide on the landscape and are not able to enter buildings anymore. The screenshots in Figure 5.4 shows how it looks like standing on the ground.

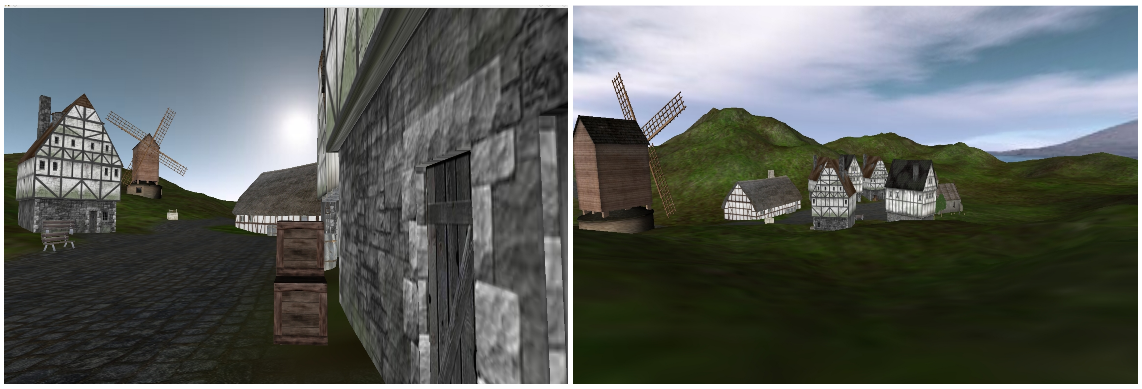


Figure 5.4: The medieval town under a blue sky

5.3 Summary

This chapter has introduced the basic concepts of collision maps and height maps in order to implement terrain following. The flexibility of the transformation management has been illustrated by the use of transformation modifiers and pipes.
Defining additional pipes and adding modifiers will become important in the subsequent chapters. In the next step we will demonstrate the interaction handling.

Chapter 6

Interaction

This part of the tutorial makes use of a modified HOMER (**H**and-centered **O**bject **M**anipulation **E**xtending **R**ay-casting) [BH97] interaction technique in order to pick and rearrange objects in the VE. Other interaction methodologies like virtual hand or GoGo [PBWI96] are implemented and provided as out-of-the box features of the framework.
Interaction in *inVRs* is rather complex since many aspects are involved. Some examples are the user, the virtual world, the transformation and event handling and if available the network. As a special feature *inVRs* is designed for concurrent object manipulation, which will we explained in a different tutorial.

6.1 State Machine

Again as with the navigation the interaction of the *inVRs* framework follows an unconventional approach. Interaction is implemented as a state machine with the three different states idle, selection, and manipulation. Figure 6.1 illustrates the state machine.

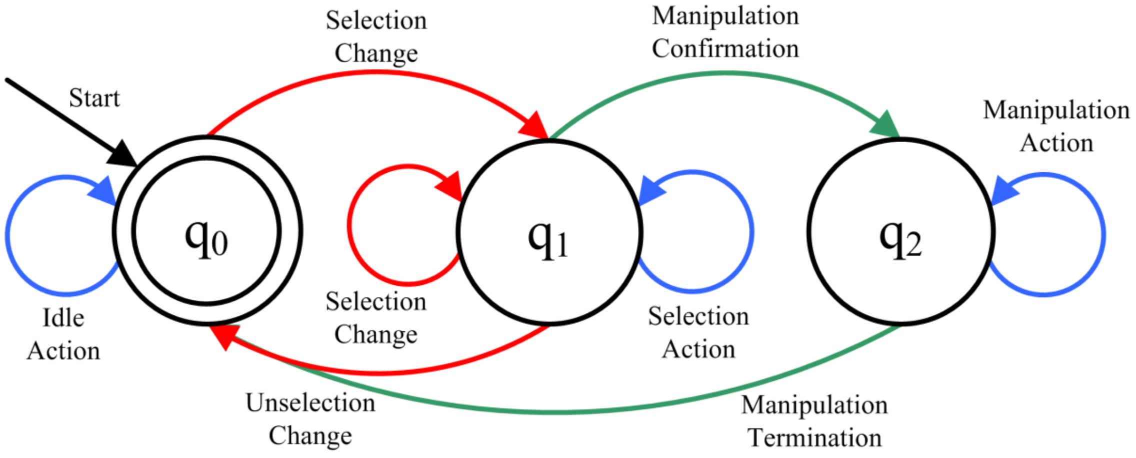


Figure 6.1: The Interaction State Machine

To switch between states transition functions are available. The states are implemented as simple enum collection, the interesting thing are really the transition functions. The states are fixed but the transition functions are designed to be exchanged with already existing or newly developed ones. The following list gives an overview on what models are available to implement the transition functions:

- `SelectionActionModel`
  This model specifies how the user is notified of the selection of an entity for example by highlighting it. Action models are executed as long as the user is in the appropriate state.
- `ManipulationActionModel`
  As the SelectionActionModel this model defines what happens to the entity during manipulation state.
- `SelectionChangeModel`
  This model is responsible for the type of selection. If the model returns true the state changes from q_0 to q_1.
- `UnselectionChangeModel`
  In case this model returns true an entity becomes unselected again and the state is changed back to idle.
- `ManipulationConfirmationModel`
  This model becomes active when we are in the Selection state. It implements the transition from selection into the manipulation state. One example for changing into the manipulation state could be simply pressing a button.
- `ManipulationTerminationModel`
  When this model returns true the transition from the manipulation into the idle state takes place. A new selection process can begin.

The different models or transition functions can be individually set up and configured in the file `interaction.xml`. This file is already prepared and does not have to be changed by you. We do not want to go into the specifics of the models but be aware, that each transition function can have its own model specific arguments.

```xml
<?xml version="1.0"?>
<!DOCTYPE interaction SYSTEM "http://dtd.inVRs.org/interaction_v1.0a4.dtd">
<interaction version="1.0a4">

  <stateActionModels>
    <selectionActionModel type="HighlightSelectionActionModel">
      <arguments>
        <arg key="modelType" type="string" value="OSG"/>
        <arg key="modelPath" type="string" value="box.osg"/>
      </arguments>
    </selectionActionModel>
    <manipulationActionModel type="HomerManipulationActionModel">
      <arguments>
        <arg key="usePickingOffset" type="bool" value="true"/>
      </arguments>
    </manipulationActionModel>
  </stateActionModels>

  <stateTransitionModels>
    <selectionChangeModel type="LimitRayCastSelectionChangeModel">
      <arguments>
        <arg key="rayDistanceThreshold" type="float" value="5"/>
      </arguments>
    </selectionChangeModel>
    <unselectionChangeModel type="LimitRayCastSelectionChangeModel">
      <arguments>
        <arg key="rayDistanceThreshold" type="float" value="5"/>
      </arguments>
    </unselectionChangeModel>
    <manipulationConfirmationModel type="ButtonPressManipulationChangeModel">
      <arguments>
        <arg key="buttonIndex" type="int" value="0"/>
      </arguments>
    </manipulationConfirmationModel>
    <manipulationTerminationModel type="ButtonPressManipulationChangeModel">
      <arguments>
        <arg key="buttonIndex" type="int" value="0"/>
      </arguments>
    </manipulationTerminationModel>
  </stateTransitionModels>

</interaction>
```

Listing 6.1: interaction.xml

When developing an interaction technique it is often common to implement a whole set of models which work nicely in composition with each other. Although they are fully exchangeable a wild combination does not make sense.

6.2 Implementing Interaction

Initially we have to make sure that the whole interaction setup is configured and registered properly. As a first step we provide the `Configuration` information where the interaction configuration is stored. As usual we edit our main `general.xml` configuration file and insert the appropriate path.

```
    <path name="InteractionModuleConfiguration"
        directory="config/modules/interaction/" />
```

Listing 6.2: Snippets4.xml - Snippet-4-1 → general.xml

Like with the `Navigation` we have to take care that the correct configuration file is loaded. The filename for our interaction configuration which we have seen in the last section has to be provided to the `SystemCore`.

```
  <module name="Interaction" configFile="interaction.xml" />
```

Listing 6.3: Snippets4.xml - Snippet-4-2 → modules.xml

And again we have to register the module in an initial callback. We are then finished with our basic module loading functionality.
This setting up might seem cumbersome, but it is straight forward and has the advantage that this code can be reused in every application. The system core provides an additional helper class, which hides a fair bit of these setting up the system configurations. For later application development it might be interesting to take a look at the `ApplicationBase`.

```
  // store the Interaction as soon as it is initialized
  else if (module->getName() == "Interaction") {
    interaction = (Interaction*)module;
  }
```

Listing 6.4: CodeFile4.cpp - Snippet-4-1 → MedievalTown.cpp

Now is the time to start with the interaction specific parts. Along with interaction techniques it often happens that the cursor is transformed in a different way than with a usual virtual hand technique thus we have to include as well a `CursorTransformationModel` which is to be set in the user configuration file `config/systemcore/userdatabase/userDatabase.xml`. The `CursorTransformationModel` describes the behavior of the users cursor during the interaction

process.
As an example, when we use tracked input devices and a virtual hand interaction technique the position and orientation of the input device is directly mapped on position and orientation of the cursor. In our case with the HOMER technique and the `HomerCursorModel`, the cursor moves towards the object when the transition from selection to manipulation takes place. If the manipulation state is left again the cursor moves back to the user.
Additionally to the `CursorTransformationModel` we also have to load a representation for the cursor so that it is rendered. This representation is defined in the configuration file entered in the cursorRepresentation element. As a visual representation of the cursor we have chosen a hand since it fits to our given scenario. A pointer or a plane might be more helpful for example for visualizations.

```
  <cursorRepresentation configFile="handRepresentation.xml" />
  <cursorTransformationModel configFile="homerCursorModel.xml" />
```

Listing 6.5: Snippets4.xml - Snippet-4-3 → userDatabase.xml

The definition of our `CursorTransformationModel` which stored inside the configuration file `homerCursorModel.xml` describes the speed of the cursor movement. The additional attributes forwardThreshold and backwardThreshold describe values which are relevant for collision detection with the object, don't worry about them now. No changes have to be applied here since it is considered a standard configuration file. Other configurations are provided for the GoGo and virtual hand interaction techniques.

```
<?xml version="1.0"?>
<!DOCTYPE cursorTransformationModel SYSTEM "http://dtd.inVRs.org/
    cursorTransformationModel_v1.0a4.dtd">
<cursorTransformationModel version="1.0a4">
  <model name="HomerCursorModel">
    <arguments>
      <arg key="animationSpeed" type="float" value="12"/>
      <arg key="forwardThreshold" type="float" value="0.1"/>
      <arg key="backwardThreshold" type="float" value="0.1"/>
    </arguments>
  </model>
</cursorTransformationModel>
```

Listing 6.6: homerCursorModel.xml

Since we are using a cursor now we have to add the appropriate modifiers in the navigation pipe of the `TransformationManager`. The cursor position in three-dimensional space of course changes with the camera position, thus these modifiers are necessary. Once the user has moved in the VE the cursor relative to the users position is taken into account and an updated transformation is written back.

```
      <modifier type="ApplyCursorTransformationModifier" />
      <modifier type="CursorTransformationWriter" />
```

Listing 6.7: Snippets4.xml - Snippet-4-4 → modifiers.xml

To finally use the interaction with our entities one new pipe is needed which maps the transformations generated in the manipulation state on the Entity which is being manipulated. In order to provide this mapping we have to open again our `modifier.xml` file and insert the following snippet. The pipe takes `Interaction` as a source and writes the `TranformationData` on the `WorldDatabase`. This should be valid for all types of relevant objects.
Lets have a brief look at the modifiers we use in our interaction pipe. The first modifier in the

pipe, the `ManipulationOffsetModifier`, is used for applying an additional picking offset. When we pick the object we the cursor moves to the center of the object. To avoid this behavior we use the offset provided by the modifier.
The second modifier in the pipe the `EntityTransformationWriter` is used to finally write the transformation of the manipulated entity in the `WorldDatabase`.

```
    <pipe srcComponentName="InteractionModule" dstComponentName="WorldDatabase"
      pipeType="Any" objectClass="Any" objectType="Any" objectId="Any"
      fromNetwork="0">
      <modifier type="ManipulationOffsetModifier"/>
      <modifier type="TransformationDistributionModifier"/>
      <modifier type="EntityTransformationWriter" />
    </pipe>
```

Listing 6.8: Snippets4.xml - Snippet-4-5 → modifiers.xml

So far this is was the setup of the transition functions, the general integration of the interaction module and the settings for the cursor transformation as well as the modifier configuration of the transformation management. There is basically one more thing to do.
When we want to select objects during interaction it is often helpful to have a representation of the cursor. We have to set again the path for the configuration of the CursorRepresentation and the `CursorTransformationModel` in the `general.xml` configuration file.

```
    <path name="CursorRepresentationConfiguration"
        directory="config/systemcore/userdatabase/cursorRepresentation/" />
    <path name="CursorTransformationModelConfiguration"
        directory="config/systemcore/userdatabase/cursorTransformationModel/" />
```

Listing 6.9: Snippets4.xml - Snippet-4-6 → general.xml

The interaction processing has to be invoked by calling the step method. During this step the transition functions are checked and updated.
Besides the interaction updates also the CursorRepresentations have to be updated. This is needed in order to allow the representations to change their look depending on the current state of the `Interaction` module. In our example the cursor changes its look when an entity is grabbed to a closed hand while it is visualized as opened hand during idle and selection state.

```
  interaction->update(dt);

  UserDatabase::updateCursors(dt);
```

Listing 6.10: CodeFile4.cpp - Snippet-4-2 → MedievalTown.cpp

When we recompile now and execute our medieval town application we should be able to pick up and drop the benches, boxes and marble balls of the VE by pressing the left mouse button. By pointing at objects with the hand representation visual feedback should be provided in form of a flashing box around the object. This is for example implemented in the `SelectionActionModel` and can be configured in the setup of the interaction.
If we release these objects they unfortunately get stuck in the air, since we do not make use of gravity. Pictures of that are provided in Figure 6.2 To overcome the problem we could create an additional modifier implementing the dropping, include the `HeightMapModifier` to transform them directly on the terrain height or alternatively we could make use of the physics module, which will be explained in another tutorial.

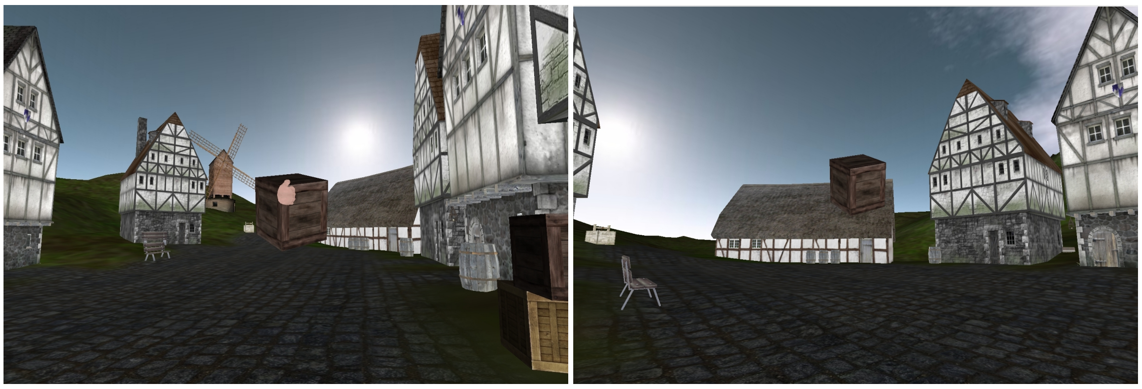


Figure 6.2: Interacting with Entities

6.3 Events

In general it is worth mentioning that the `Interaction` makes use of the `SystemCore`s internal event system the `EventManager`. State changes of the automaton are for example propagated throughout the system, most of them are automatically distributed via the network in case this module is present as well. Understanding in the *inVRs* `Event`s is not relevant for the tutorial, but it is a must look-up if you want to develop your own interaction techniques.

6.4 Summary

This chapter has given a brief overview on the interaction concept of the *inVRs* framework. A state machine was configured with a set of transition functions in order to implement a modified HOMER technique. A cursor transformation model was introduced to move a representation of the cursor to the entity which should be manipulated. Additionally a new pipe was specified to implement object manipulation. The users are able to pick up entities like boxes or benches in the VE.
In the next chapter we will try to interact with several users in a shared VE. All participants should then be able to observe navigating and interacting users.

Chapter 7

Using Network Communication

Multi-user environments can be more exciting than simple single user applications. And why would you need a whole town if you only have one inhabitant. In the following the `Network` module is introduced which will allow us to share a virtual world. Navigation and interaction will be distributed.

7.1 Concepts

Many approaches for developing large scale networked virtual environments exist. An ideal solution is impossible to find since special aspects like consistency and response time are ambivalent. Therefore many implementations of this module exist using an internal `NetworkInterface` between the core and the module to keep the exchangeability.
Early drafts of the Network module included zoning mechanism to distribute the virtual world over several servers [AHHV05, AHHV04, AHV04]. An example for the exchangeability of the *inVRs* network module was demonstrated when an inVRs application was ported to use GRID infrastructures [ALBV08].
The *inVRs* `Network` module is the most likely module to be completely user-implemented. Many different requirements like scalability or response times are often completely application specific. As long as the developer of the network module sticks to the interface to the modules and the core basically any communication and data distribution topology can be implemented.
The default implementation of the module can be considered robust and straight forward. It used replicated databases and one-to-all communication mechanisms.

7.2 Setting up the Network Communication

Let's start with our usual module setup and add the path of the network module configuration file in the `general.xml` configuration.

```
    <path name="NetworkModuleConfiguration"
        directory="config/modules/network/" />
```

Listing 7.1: Snippets5.xml - Snippet-5-1 → modules.xml

And again we have to set the filename of the configuration and define the name of the library in order to load it dynamically.

```
  <module name="Network" configFile="network.xml" />
```

Listing 7.2: Snippets5.xml - Snippet-5-2 → modules.xml

Of course we have to register again a callback for supporting the plugin mechanism of the framework.

```
  // store the NetworkInterface as soon as it is initialized
  else if (module->getName() == "Network") {
    network = (NetworkInterface*)module;
  }
```

Listing 7.3: CodeFile5.cpp - Snippet-5-1 → MedievalTown.cpp

Now we can start with integrating our network code and setting the network specific configurations. The first code snippet is used for connection establishment. The command line parameter which should an IP address or a hostname is the machine to connect to. Additionally we have to specify the port to connect to, separated from the port by a colon.
The `NetworkInterface` :: `connect()` method tries to establish a connection to the passed address. Afterwards we have to call the synchronize method of the system core in order to update the databases.

```
  // try to connect to network first command line argument is {hostname|IP}:port
  if (argc > 1) {
    printf("Trying to connect to %s\n", argv[1]);
    network->connect(argv[1]);
  }
  SystemCore::synchronize();   // synchronize both VEs
```

Listing 7.4: CodeFile5.cpp - Snippet-5-2 → MedievalTown.cpp

The `SystemCore` has to be triggered at the beginning of the display loop in order to process the event handling.

```
  SystemCore::step();  //update the system core, needed for event handling
```

Listing 7.5: CodeFile5.cpp - Snippet-5-3 → MedievalTown.cpp

The ports for UDP and TCP communication are set. The basic implementation of the network module transmits `Events` via TCP and `TransformationData` via UDP. User defined messages can be distributed as well with additional methods.

```xml
<?xml version="1.0"?>
<!DOCTYPE network SYSTEM "http://dtd.inVRs.org/network_v1.0a4.dtd">
<network version="1.0a4">
  <ports TCP="8081" UDP="8082"/>
</network>
```

Listing 7.6: network.xml

7.3 Transmitting Data

Besides the specific synchronization and connection establishment data, *inVRs* transmits `Events` and `TransformationData`.
The `EventManager` automatically detects whether the network module is present, if this is the case, the events are distributed to the remote participant, unless specified differently.

The data of the TransformationManager has to be sent in a different way. In order to transmit the transformations the TransformationDistributionModifier has to be included in the `modifiers.xml` file. In our case this has to happen at two areas in the configuration, once for the Navigation and once for the Interaction.

```
      <modifier type="TransformationDistributionModifier" />
```

Listing 7.7: Snippets5.xml - Snippet-5-3 → modifiers.xml

Two additional pipes have to be created in order to react on remote transformations coming in. The WorldDatabase has to write the transformations of the remote Interaction modules as well. Otherwise we would have a slightly inconsistent and rather static shared VE. In this snippet the attribute *fromNetwork* is set to 1. meaning the data has to be sent from the Network module to the TransformationManager.

```
    <pipe srcComponentName="InteractionModule" dstComponentName="WorldDatabase"
      pipeType="Any" objectClass="Any" objectType="Any" objectId="Any"
      fromNetwork="1">
      <modifier type="EntityTransformationWriter" />
    </pipe>
```

Listing 7.8: Snippets5.xml - Snippet-5-4 → modifiers.xml

Of course the transformation of the remote users has to be transmitted as well. Thus an additional pipe has to be specified in order to represent any remote users.

```
    <pipe srcComponentName="NavigationModule" dstComponentName="
        TransformationManager"
      pipeType="Any" objectClass="Any" objectType="Any" objectId="Any"
      fromNetwork="1">
      <modifier type="UserTransformationWriter" />
      <modifier type="AvatarTransformationWriter" >
        <arguments>
          <arg key="clipRotationToYAxis" type="bool" value="true" />
        </arguments>
      </modifier>
    </pipe>
```

Listing 7.9: Snippets5.xml - Snippet-5-5 → modifiers.xml

By having the transformation and event distribution decoupled from the navigation and the interaction, it is easily possible to use different interaction and navigation techniques on the interconnected sites. This can for example become important if CAVE users like to interact with desktop users in the same NVE.

7.4 Displaying Avatars

Now you can recompile and try to connect to the IP address of your neighbor. You should see your remote partner now since their avatars have been previously set in the UserDatabase they are configured via an intuitive XML description in the file `config/systemcore/userdatabase/avatar/avatar.xml`.
More complex avatars are provided by *inVRs* as well as an additional tool. It is possible to set animation cycles on the or move specific parts of their bodies, which becomes interesting when tracking systems are used with *inVRs*.

```
<?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 7.10: avatar.xml

We have now completed the network chapter. Since the transformations are distributed the movement of the avatars should be visible as well as the interaction they perform.

7.5 Execution

For testing the application over the network first start an instance of the tutorial on one machine using the `startTutorial`-script. As soon as the application is running you can connect from another machine to the running application by passing the hostname or ip-address and the tcp-port to the start-script separated by a colon, e.g.:

```
./startTutorial.sh 192.168.0.100:8081
```

NOTE: if you want to start multiple application instances on the same machine you will have to use different TCP and UDP ports. Therefore you will have to use a separate `network.xml` configuration file for every instance.
Figure 7.1 shows two users in one virtual environment. The small pictures display the view of the user interacting with an entity and the big ones show what an observing user sees.

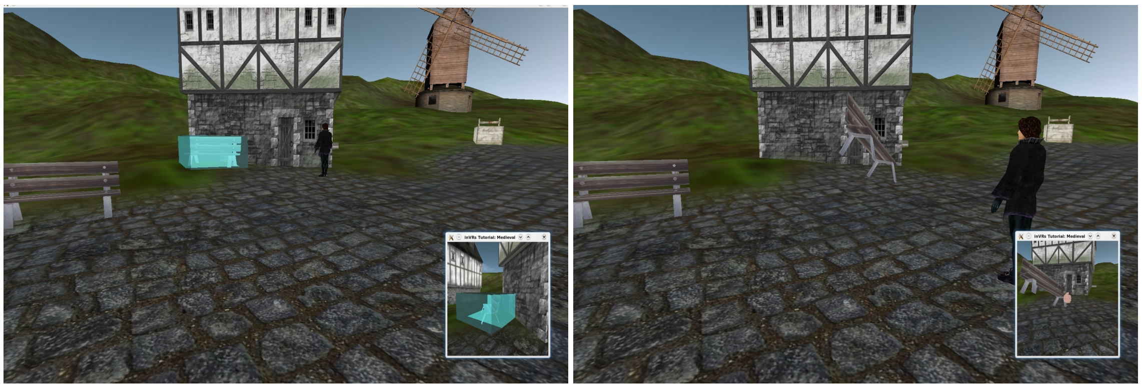


Figure 7.1: Interacting in a networked virtual environment

7.6 Summary

We should now be able to share the same virtual world with other participants. Most of the data distribution is hidden fully from the application developer. The distribution mechanism with the help of the `EventManager` and the `TransformationManager` hide the details from application development. If you have written a single user VE with the help of *inVRs* it should be fairly straight forward to port it into the domain of NVEs.

Chapter 8

Developing own Application Logic

One of the main advantages which *inVRs* provides compared to other systems in the field is the full flexibility given to the application developer which will be illustrated in the following.
In this example we are going to rotate the sails of the windmill by writing our own animation code. To achieve this we have to gather input for starting and stopping the animation. Additionally we have to transform the received input into the rotating behavior of the windmill. This access on the windmill sails has to happen on a lower level than maybe expected.

8.1 Input and Animations

When we take a look at the following snippet, we can basically identify two sections in the code. The first one is processing the input at the top part of the snippet and the second one at the bottom is used for implementing the animation and writing it back to the entity.
To implement our animation, we evaluate the right mouse button, which is mapped internally as a button of our `Controller` object which carries the id 2 as defined in the file `config/interfaces/controllermanager/MouseKeybController.xml`. If this button is pressed the variable windMill-Speed, describing the rotational speed of the sails, will be increased based on the time difference between the last measurements. In case the windMillSpeed raises above a defined threshold (2π), the speed will be limited to the threshold. If we release the button the speed is decremented until it reaches the lower threshold of 0.

```
if (controller->getButtonValue(2)) { // the right mouse button is pressed
  windMillSpeed += dt*0.5;           // increase speed of the windmill
  if (windMillSpeed > 2*M_PI) {
    windMillSpeed = 2*M_PI;
  }
} else if (windMillSpeed > 0) {      // pressing mouse button stopped
  windMillSpeed -= dt*0.5;           // decrease speed of windmill
} else if (windMillSpeed < 0) {
  windMillSpeed = 0;
}
```

Listing 8.1: CodeFile6.cpp - Snippet-6-1 - Top Part → MedievalTown.cpp

The bottom section of the snippet is used for finally implementing the animation. If the rotational speed is above 0 the sails are to be animated. At first we have to request the `Entity` of the from our `WorldDatabase` by calling the `WorldDatabase::getEntityWithEnvironmentId()` function which takes two parameters. The first one is the id of the `Environment` the `Entity` is in, the second one is the id of the `Entity` which was specified was previously specified in `environment.xml`. In general other possibilities to look up entity, as for example by name are possible as well.
We don't want to operate on an entity basis because we do not want to rotate the whole windmill.

As a result we will have to dig lower in the entity and access it on a scene graph level. Thus we create a `ModelInterface` and retrieve the sub scene graph from the model which is stored in our our entity by grabbing its to scene graph node. We now have to make sure that it is a transformation node and cast it as `TransformationSceneGraphNodeInterface`.
Now it is time to update the `TransformationData` of the windmill sails. We have to use a bit of the math function provided by the GMTL. Let's first create a quaternion [1] based on its AxisAngle attribute. As an axis we simply set the z-axis, since this is the one we want to rotate the sails around. The angle is based on the windMillSpeed variable which was previously set in the interaction part of the code snippet.
After having calculated the current rotation change based on the speed and the time passed we have to multiply it with the already present rotation of the sails. Watch out we are working in this case with `TransformationData` not with matrices, so we are only changing the orientation attribute of the transformation data and not the translation or scale attributes. Finally the newly calculated transformation has to replace the current transformation in the transformation node via the `TransformationSceneGraphNodeInterface::setTransformation()` method.

```
if (windMillSpeed > 0) {            // rotate sails
  // retrieve the windmill entity
  Entity* windMill = WorldDatabase::getEntityWithEnvironmentId(1, 27);
  ModelInterface* windMillModel = windMill->getVisualRepresentation();

  // retrieve the windmill's sails
  SceneGraphNodeInterface* sceneGraphNode =
    windMillModel->getSubNodeByName("Sails");

  // make sure this node is a transformation node
  assert(sceneGraphNode->getNodeType() ==
    SceneGraphNodeInterface::TRANSFORMATION_NODE);
  TransformationSceneGraphNodeInterface* transNode =
    dynamic_cast<TransformationSceneGraphNodeInterface*>(sceneGraphNode);
  assert(transNode);

  // rotate the sails
  TransformationData trans = transNode->getTransformation();
  gmtl::AxisAnglef axisAngle(windMillSpeed*dt, 0, 0, 1);
  gmtl::Quatf rotation;
  gmtl::set(rotation, axisAngle);
  trans.orientation *= rotation;
  transNode->setTransformation(trans);
}
```

Listing 8.2: CodeFile6.cpp - Snippet-6-1 - Bottom Part → MedievalTown.cpp

Now we should recompile and execute the application again. If we keep the right mouse button pressed the wheel of the windmill is going to start rotating, when we release it again it will slow down till it stops. This is shown in Figure 8.1 Of course this was just a simple example but illustrates quite well how flexible the approach is.
An different option for creating the animation could have been an implementation making use of the `TransformationManager`. In such a case it would have been easily possible to distribute the rotation via modifier, to the other participants. Since we have not made use of the distribution mechanisms, we will only have local changes in our NVE.

[1] Quaternions are often used to describe rotations, basically they have an axis and an angle to rotate around the axis. If you want to understand them in detail have a look at Shoemakes' paper [Sho85]

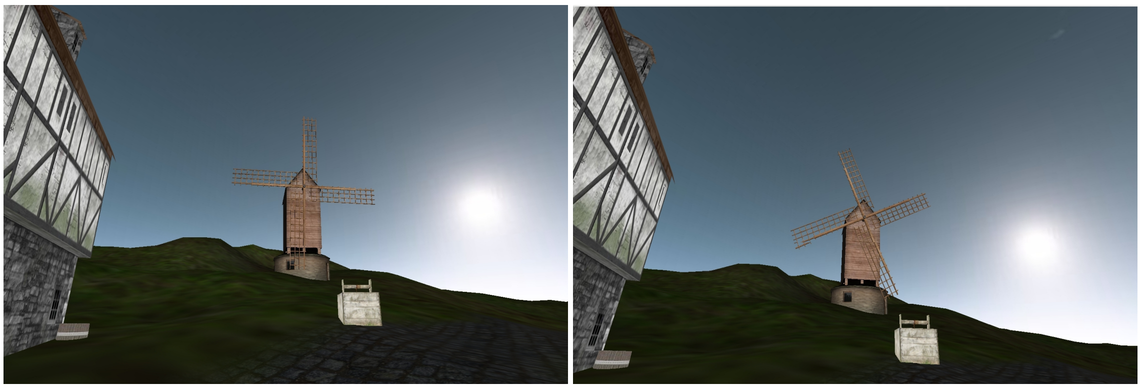


Figure 8.1: A spinning windmill wheel

8.2 Summary

We are now finished with the tutorial and have learned in this chapter how to develop own application logic. Input from the controller was polled and evaluated to generate a speed value. Afterwards an entity was requested from the world database and accessed on a lower level in order to manipulate its sub scene graph.
In overall we already created a powerful application compared to the code which had to be written. There is much much more what you can do with *inVRs* and some additional features and tools will be explained in the final chapter.
Well, the next step would be porting this tutorial on a stereoscopic [2] system with tracked input devices. You are way not as far away from this as you might think.
By integrating the CAVESceneManager and configuring it as well as your devices, you will still have to recompile, but you will only have to change a small amount of code. The main work is in that case setting up the display which uses a similar definition like the CAVELib, where configuration might be available already.

[2]maybe even a multi-display system like a CAVE

Chapter 9

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.

9.1 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 value 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.

9.2 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 value 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`.

9.3 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 9.1: 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 9.2: 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 9.3: 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 9.4: 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 9.5: 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 9.6: 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 9.7: 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, but please remember to insert your plugin path into general.xml, like you have done in the Medieval Town application:

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

Listing 9.8: general.xml - Enter Path to inVRs Libraries

When executing this application you will see a predefined scene containing an plane, a coordinate system and the *inVRs* logo. This is shown in Figure 9.1. Due to the configured navigation module you can navigate through this environment via mouse and keyboard input.

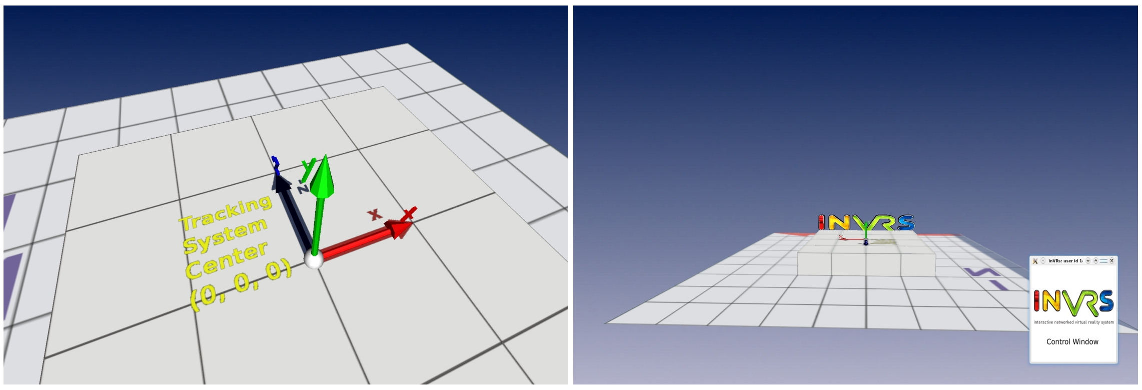


Figure 9.1: The basic Going Immersive application

9.4 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 10

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.

10.1 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 10.1 illustrates the most prominent VR displays, a CAVE on the left side and an HMD on the right side.

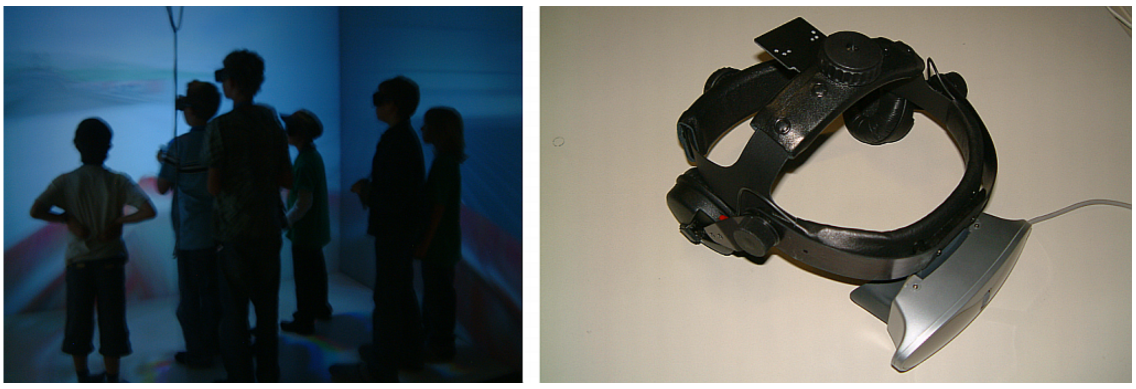


Figure 10.1: A CAVE and an HMD

[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 10.2 show two wide spread displays which are commonly used for larger audiences, where the previously introduced displays are more applicable for single users.

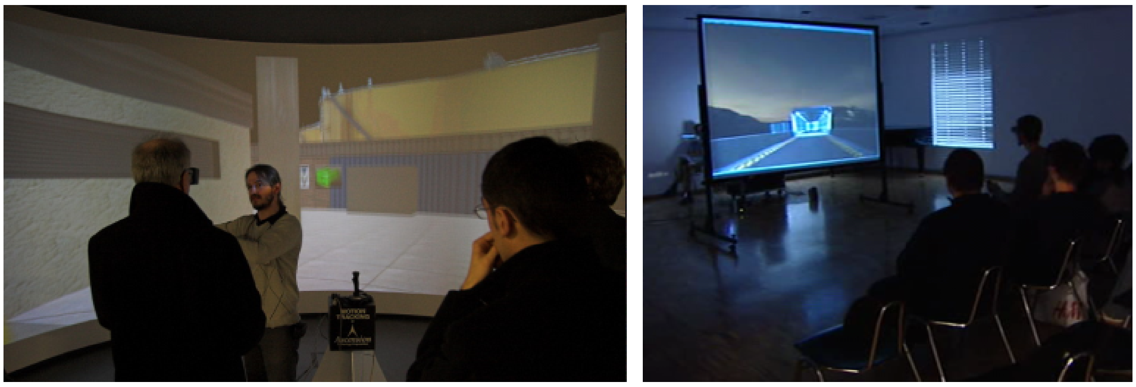


Figure 10.2: A Curved Screen and a Powerwall

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.

10.2 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 SimpleScene-Manager.

- `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`

10.3 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 10.1: GoingImmersive.cpp

10.4 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 10.2: 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 10.3: 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 10.4: 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 10.5: CodeSnippets1.cpp - Snippet-1-1 → GoingImmersive.cpp

When building and starting the application now you should see the same result as shown in Figure 10.3:

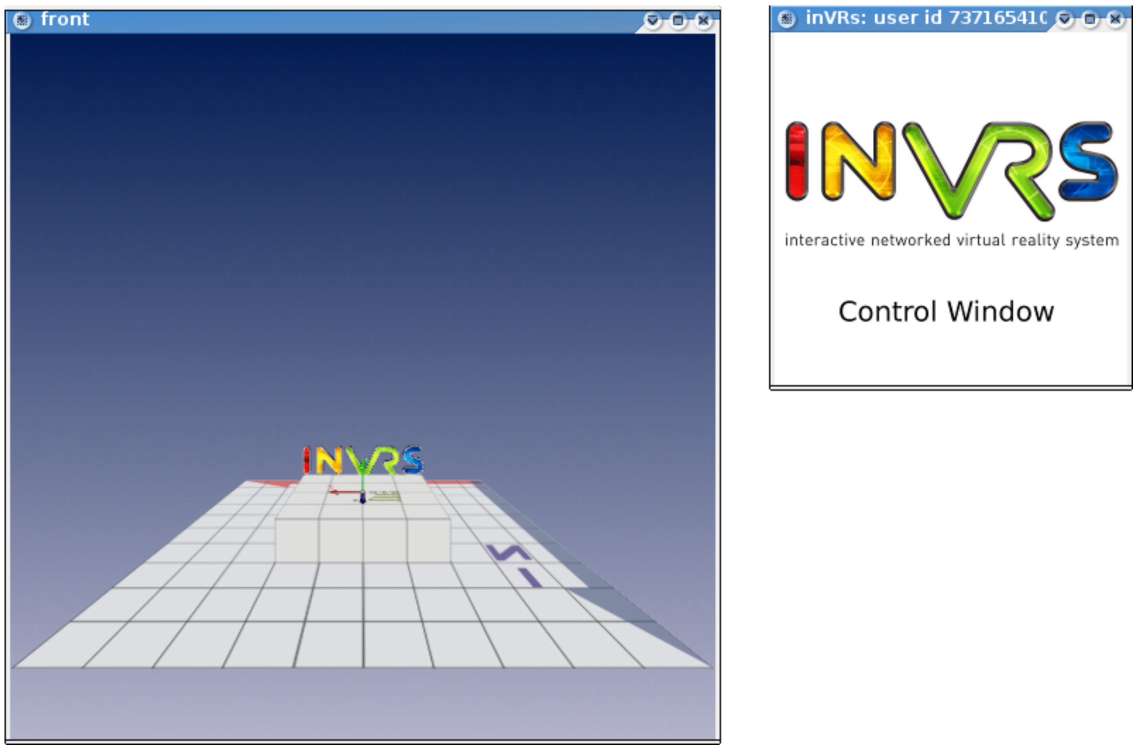


Figure 10.3: Render server window (front) and control window (inVRs)

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.

10.5 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 11

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.

11.1 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 11.1: Some Typical VR Input devices

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

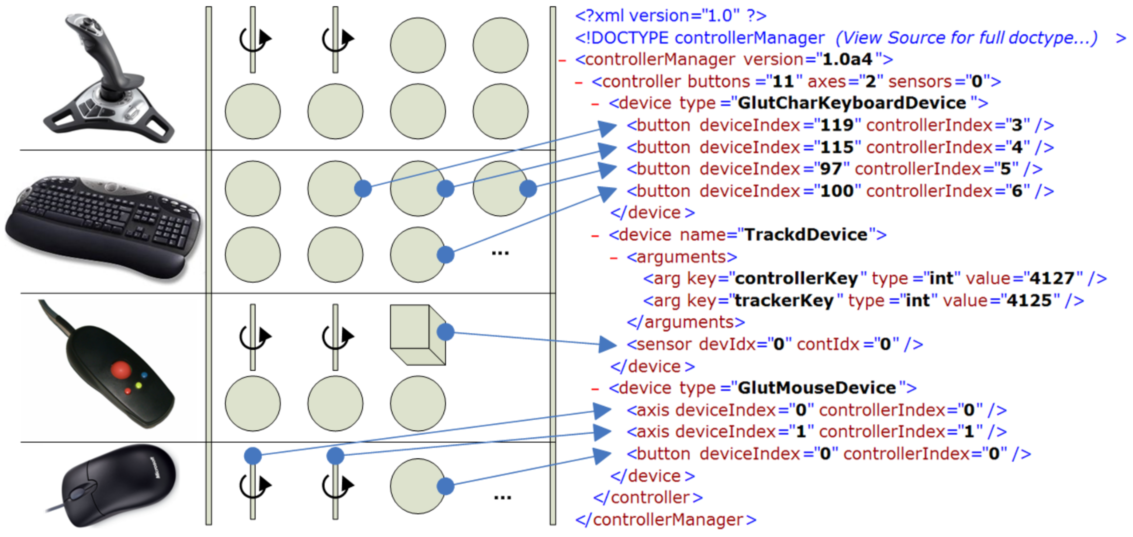
Figure 11.2 shows some devices which are wide spread in the field of Virtual Reality. On the left side a space mouse, emulating a 6 DOF sensor is shown, the middle illustrates a wand with buttons and joystick and the right side of the figure shows a pair of pinch gloves which generate boolean values on contact of the finger tips.

11.2 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.



```
<?xml version="1.0" ?>
<!DOCTYPE controllerManager (View Source for full doctype...) >
- <controllerManager version="1.0a4">
  - <controller buttons ="11" axes="2" sensors="0">
    - <device type ="GlutCharKeyboardDevice ">
        <button deviceIndex="119" controllerIndex="3" />
        <button deviceIndex="115" controllerIndex="4" />
        <button deviceIndex="97" controllerIndex="5" />
        <button deviceIndex="100" controllerIndex="6" />
      </device>
    - <device name="TrackdDevice">
      - <arguments>
          <arg key="controllerKey" type ="int" value="4127" />
          <arg key="trackerKey " type ="int" value="4125" />
        </arguments>
        <sensor devIdx="0" contIdx ="0" />
      </device>
    - <device type ="GlutMouseDevice ">
        <axis deviceIndex="0" controllerIndex="0" />
        <axis deviceIndex="1" controllerIndex="1" />
        <button deviceIndex="0" controllerIndex="0" />
      </device>
    </controller>
</controllerManager>
```

Figure 11.2: 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 11.2. 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.

11.3 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 11.1: 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 11.2: 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 11.3: 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 11.4: 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 11.5: 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 11.6: 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 11.7: VrpnDevice.cpp - accessor methods for input data

Now with these 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 obtained 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 11.8: 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.

11.4 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 12. 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 11.9: 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 11.10: 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 11.11: 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 11.12: 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
<?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 11.13: wandNavigation.xml

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

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

Listing 11.14: 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.

11.5 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 12

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 12.1: 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.

12.1 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.

12.2 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
<?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 12.2: 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.

12.3 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 12.3: CodeSnippets3.cpp - Snippet-3-1 → GoingImmersive.cpp

Next the factory is registered in the `WorldDatabase`. This is done in the `initCoreComponentCallback`() 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 12.4: 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 12.5: 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 12.6: 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 12.7: CodeSnippets3.cpp - Snippet-3-4 → GoingImmersive.cpp

In the `initialize()` method 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 12.8: 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 12.9: 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 12.10: 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.

12.4 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
<?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 12.11: 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. Figure 12.1 shows what you will see.

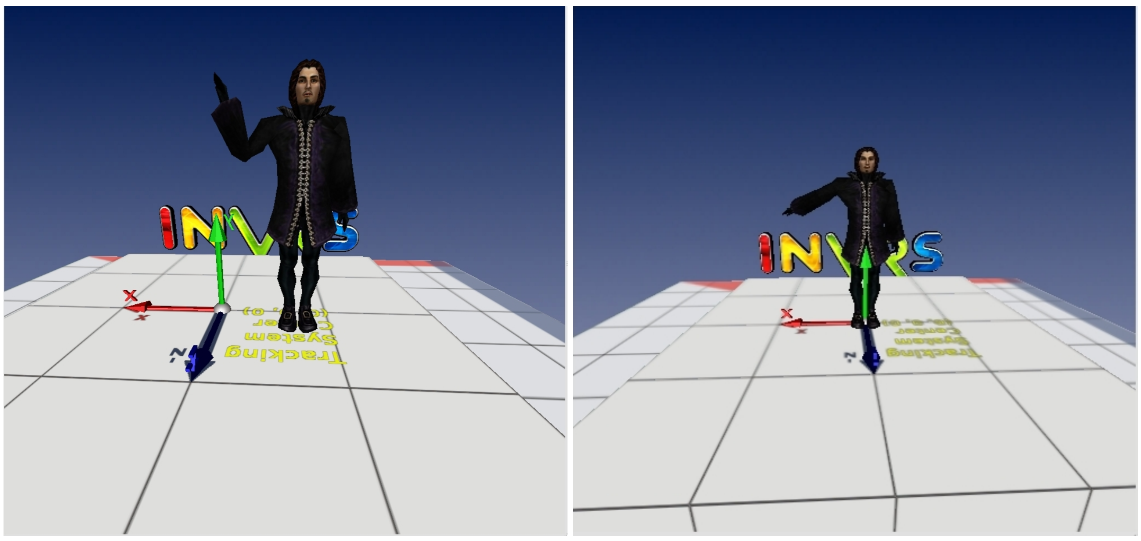


Figure 12.1: The Avatara avatar

12.5 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 13

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

13.1 User Coordinates

For interaction, navigation or simply to display an avatar several coordinate systems are necessary in the *inVRs* framework. Figure 13.1 illustrates the most important *inVRs* coordinate systems.

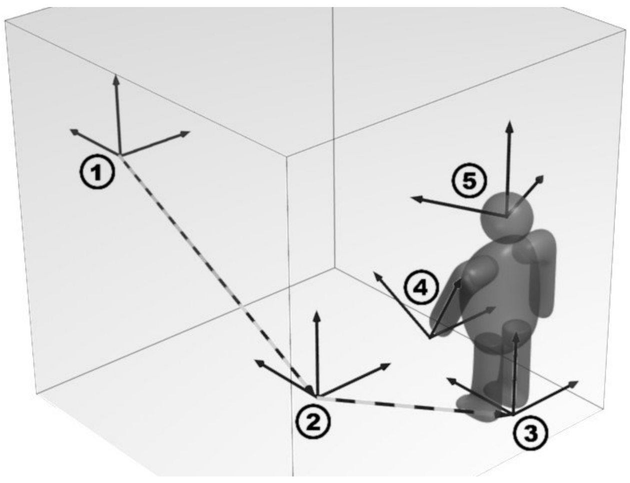


Figure 13.1: Coordinate Systems of the User

As illustrated in Figure 13.1 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:

$$\begin{aligned} userHandTrans &= trackedUserTrans^{-1} * trackedHandTrans \\ worldHandTrans &= navigatedTrans * trackedHandTrans \end{aligned}$$

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:

$$\begin{aligned} userHeadTrans &= trackedUserTrans^{-1} * trackedHeadTrans \\ worldHeadTrans &= navigatedTrans * trackedHeadTrans \end{aligned}$$

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:

$$\begin{aligned} & i...sensorIndex \\ userSensorTrans_i &= trackedUserTrans^{-1} * trackedSensorTrans_i \\ worldSensorTrans_i &= navigatedTrans * trackedSensorTrans_i \end{aligned}$$

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.

13.2 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 13.1: CodeSnippets4.cpp - Snippet-4-1 → GoingImmersive.cpp

In the `initialize()` method at first the used controller is requested from the `ControllerManager`. 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 13.2: 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 13.3: 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 13.4: 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`.

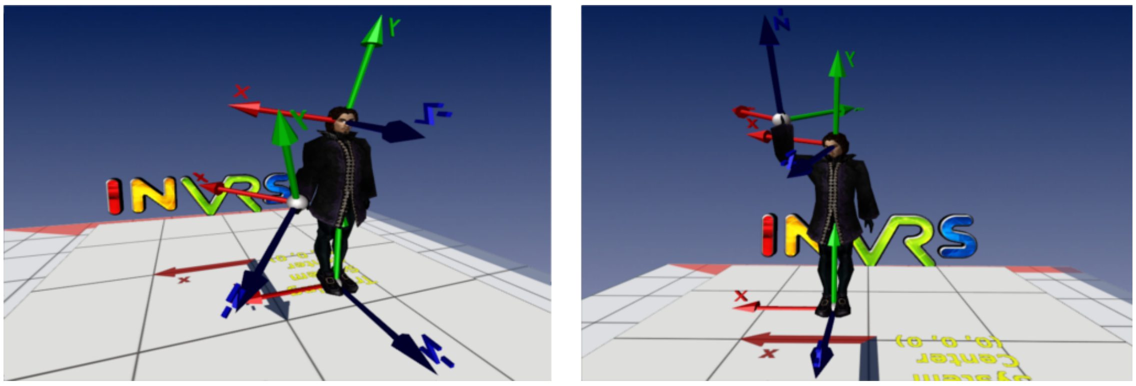


Figure 13.2: The Final Going Immersive Application

13.3 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 14

Outlook

You have seen how to create an interactive NVE using OpenSG and *inVRs*. The medieval town application has demonstrated how a fully functional application can be developed with less than 500 lines of C++ code and variety of XML configurations. By keeping the full flexibility and control over the application this approach cannot only be used for rapid prototyping but rather to create complex NVEs.
It is possible of course to enhance the framework or the developed application for example by creating own navigation and interaction models. Through the free exchange of these components it is possible to use constantly developed new features features without the need of recompiling the application and just updating the framework.
The second part of the tutorial has shown how to use the *inVRs* framework with a variety of input devices an displays. Wrapping functionality for faster application development was introduced and basic setup as described in the first part of the 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 configure and use inVRs to create networked virtual environments. Further it was shown how to interconnect arbitrary multi-display systems and input devices in order to create fully immersive NVEs. 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 control 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.
This tutorial has introduced only a few of the *inVRs* functionalities, but there is much more to explore. Besides the provided core features of the `SystemCore` and the `Navigation`, `Interaction`, and `Network` modules and the introduced external Avatara package a huge variety of different modules and tools exist.

14.1 Tools

There are many. A large variety of tools exist which accompany the *inVRs* framework. These tools are constantly made available and documented on the *inVRs* webpage.
The can be basically grouped in three different categories:

- Scene Graph Specific Tools
- External Tools
- Modules

The scene graph specific tools deal with fluid dynamics, particle systems [Ree83], free-form deformation [SP86], 3D menus as well as the introduced skybox, collision maps and height maps. An extremely valuable tool for developing immersive multi-display applications is the CAVESceneManager [HJAA05] which has been introduced as well during the tutorial. The articulated Avatars described in the second part of the tutorial are available as a standard OpenGL API, for OpenSG and have specific *inVRs* support.
As an external tool a collaborative editor has been created, which allows the intuitive layout of a VE. Using a GUI it is possible to create environments and freely arrange entities and tiles inside these environments. Currently the editor is re-wirtten and re-desigend to support future extensions of the *inVRs* framework.
The modules which have been developed so far cover 2D Physics, 3D Physics [ALV07], and animations. The free-form deformation tool can be loaded as a module as well and supports automatic network communication when objects are deformed.

14.2 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.

14.3 Documentation

The *inVRs* framework provides constantly updated documentation of the framework itself and the additional tools. Currently an additional manual describing the configuration and the XML setup of the main components is available. Additionally doxygen code documentation can be found under http:
doxygen.invrs.org. With the next releases of the framework a programmers guide will be made available which clearly describes the usage and the extension of the single core components.

14.4 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.

[AHHV04] Christoph Anthes, Paul Heinzlreiter, Adrian Haffegee, and Jens Volkert. Message traffic in a distributed virtual environment for close-coupled collaboration. In *International Conference on Parallel and Distributed Computing Systems (PDCS '04)*, pages 484–490, San Francisco, CA, USA, September 2004. ISCA.

[AHHV05] Christoph Anthes, Adrian Haffegee, Paul Heinzlreiter, and Jens Volkert. A scalable network architecture for closely coupled collaboration. *Journal of Computing and Informatics (CAI)*, 1(24):31–51, August 2005.

[AHKV04] Christoph Anthes, Paul Heinzlreiter, Gerhard Kurka, and Jens Volkert. Navigation models for a flexible, multi-mode vr navigation framework. In *ACM SIGGRAPH on Virtual Reality Continuum and Its Applications in Industry (VRCAI '04)*, pages 476–479, Singapore, June 2004. ACM Press.

[AHV04] Christoph Anthes, Paul Heinzlreiter, and Jens Volkert. An adaptive network architecture for close-coupled collaboration in distributed virtual environments. In *ACM SIGGRAPH on Virtual Reality Continuum and Its Applications in Industry (VRCAI '04)*, pages 382–385, Singapore, June 2004. ACM Press.

[ALBV07] Christoph Anthes, Roland Landertshamer, Helmut Bressler, and Jens Volkert. Managing transformations and events in networked virtual environments. In *ACM International MultiMedia Modeling Conference (MMM '07)*, volume 4352 of *Lecture Notes in Computer Science (LNCS)*, pages 722–729, Singapore, January 2007. Springer.

[ALBV08] Christoph Anthes, Roland Landertshamer, Hemut Bressler, and Jens Volkert. Developing vr applications for the grid. In *European Conference on Parallel and Distributed Computing (Euro-Par '08)*, Las Palmas de Gran Canaria, Spain, August 2008. Springer.

[ALV07] Christoph Anthes, Roland Landertshamer, and Jens Volkert. Physically–based interaction for networked virtual environments. In Yong Shi, Geert Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *International Conference on Computational Science (ICCS '07)*, volume 4488 of *Lecture Notes in Computer Science (LNCS)*, pages 776–783, Beijing, China, May 2007. Springer.

[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.

[AWL$^+$07] Christoph Anthes, Alexander Wilhelm, Roland Landertshamer, Helmut Bressler, and Jens Volkert. Net'?O'?Drom – An Example for the Development of Networked Immersive VR Applications. In Yong Shi, Geert Dick van Albada, Jack Dongarra, and

Peter M. A. Sloot, editors, *International Conference on Computational Science (ICCS '07)*, volume 4488 of *Lecture Notes in Computer Science (LNCS)*, pages 752–759, Beijing, China, May 2007. Springer.

[BH97] Douglas A. Bowman and Larry F. Hodges. An evaluation of techniques for grabbing and manipulating remote objects in immersive virtual environments. In *ACM Symposium on Interactive 3D Graphics (SI3D '97)*, pages 35–38, Providence, RI, USA, April 1997. ACM Press.

[BLAV06] Helmut Bressler, Roland Landertshamer, Christoph Anthes, and Jens Volkert. An efficient physics engine for virtual worlds. In *medi@terra '06*, pages 152–158, Athens, Greece, October 2006.

[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.

[PBWI96] Ivan Poupyrev, Mark Billinghurst, Suzanne Weghorst, and Tadao Ichikawa. The go-go interaction technique: Non-linear mapping for direct manipulation in vr. In *ACM Symposium on User Interface Software and Technology (UIST '96)*, pages 79–80, Seattle, WA, USA, November 1996. ACM Press.

[Ree83] William T. Reeves. Particle systems - a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2(2):93–108, April 1983.

[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.

[SG02] Andreas Simon and Martin Göbel. The i-cone - a panoramic display system for virtual environments. In *Pacific Conference on Computer Graphics and Applications (PG '02)*, pages 3–7, Beijing, China, October 2002. IEEE Computer Society.

[Sho85] Ken Shoemake. Animating rotations with quaternion curves. In Pat Cole, Robert Heilman, and Brian A. Barsky, editors, *International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '85)*, pages 245–254, July 1985.

[SP86] Thomas W. Sederberg and Scott R. Parry. Free-form deformation of solid geometric models. *ACM SIGGRAPH Computer Graphics*, 20(4):151–160, August 1986.

[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: gerzi-3d-art
Model: chapel.zip, windmill.zip
Source: http://www.turbosquid.com/

Author: medievalworlds
Model: fw65_lowpoly.zip, fw43_lowpoly.zip, well4.zip
Source: http://www.turbosquid.com/

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

Author: amethyst7@gotdoofed.com
Skybox: LostAtSea
Source: http://amethyst7.gotdoofed.com/

Installation Instructions

These installation instructions describe how to generate the platform specific build files (e.g. Visual Studio Projects or Unix Makefiles) using the CMake program. The instructions apply to both tutorial applications (MedievalTown and GoingImmersive).

Prerequisites

To be able to build the tutorial applications (MedievalTown and GoingImmersive) inVRs has to be built and installed. For downloading the latest source distribution of inVRs please visit the inVRs homepage at:

- http://trac.invrs.org/wiki/inVRsInstallation

On this page you can also find a prebuilt package of the latest inVRs version for Visual Studio 2005 which includes also all libraries which are required by inVRs (also OpenSG). Note that this version is not compatible with other Versions of Microsoft Visual Studio. For the installation of the prebuilt version simply download it from the homepage and unzip it somewhere on your harddrive (e.g. on C:\inVRs).
For details on building inVRs from source please have a look at the inVRs installation instructions which are also available on the inVRs homepage.
The following packages must be installed for building inVRs (as well as the tutorial applications):

- OpenSG - http://www.opensg.org
- CMake - http://www.cmake.org

Additional Prerequisites for Microsoft Windows

For building the tutorial applications on a Windows platform you need either Microsoft Visual Studio 2003 or Microsoft Visual Studio 2005. Other versions of Visual Studio won't work because the OpenSG version used by inVRs is not available for these IDEs.
These instructions were tested with following Packages:

- Microsoft Visual C++ 2005 Express
- Microsoft Windows Server 2003 R2 Platform SDK

For instructions how to configure Visual C++ for Platform SDK compatibility see:

- http://msdn.microsoft.com/en-us/library/ms235626(VS.80).aspx

Step 1 (OPTIONAL): Configure Application

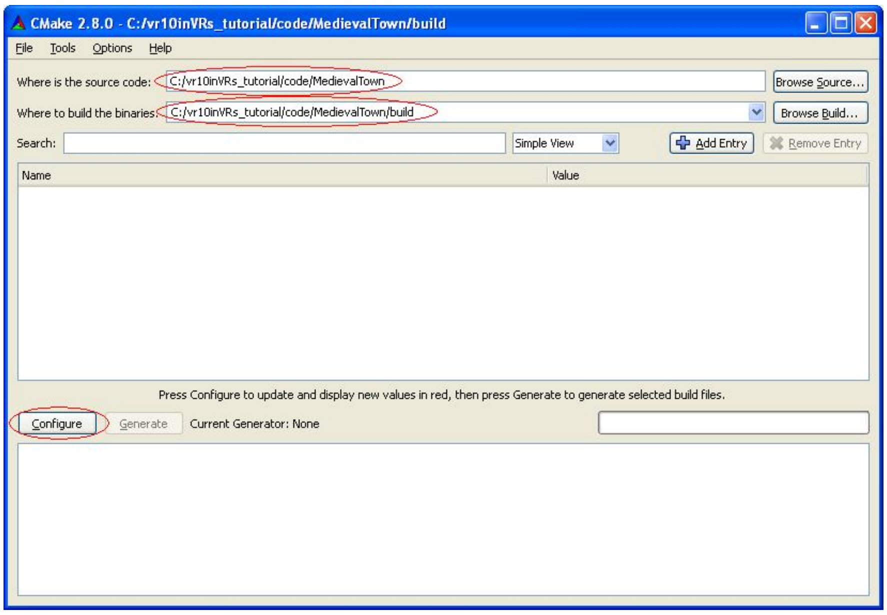
For the generation of the build files CMake needs to know where your inVRs installation can be found on your local hard drive. In order to simplify the search process for this folder you can specify this folder in the file `user.cmake`:

```
# 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 C:/inVRs)
```

Please take care to either use a single slash ("/") or two backslash signs ("\\") as separators, because a single backslash causes cmake to expect an escape sequence. Additionally you need to place an escape sign before each space-character in your folder names (if you have some), e.g. *C:/Program\ Files/inVRs*

Step 2: Generate build files with CMake-GUI

These instructions require the CMake-GUI to be available on your system. Current CMake versions provide the cmake-GUI application for most supported platforms. If you don't have the GUI version available you can also use the console tool *ccmake* or you can call the *cmake* command directly.
First open the CMake GUI and set the paths to the Tutorial project:



Where is the source code: C:/vr10inVRs_tutorial/code/MedievalTown

Where to build the binaries: C:/vr10inVRs_tutorial/code/MedievalTown/build

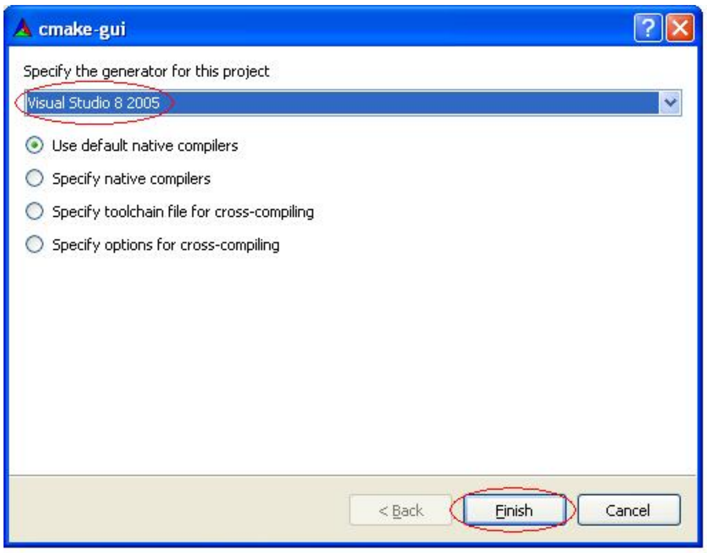
Search:

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: None

Figure 14.1: Open CMake and set Tutorial paths

Press Configure and select as target build type the desired build system. On Microsoft Windows for example you may use Microsoft Visual Studio 2005 (or Microsoft Visual Studio 2003, depending on your used IDE), on Linux or Mac OSX you should use Unix Makefiles.



Specify the generator for this project

Visual Studio 8 2005

[✓ Use default native compilers]

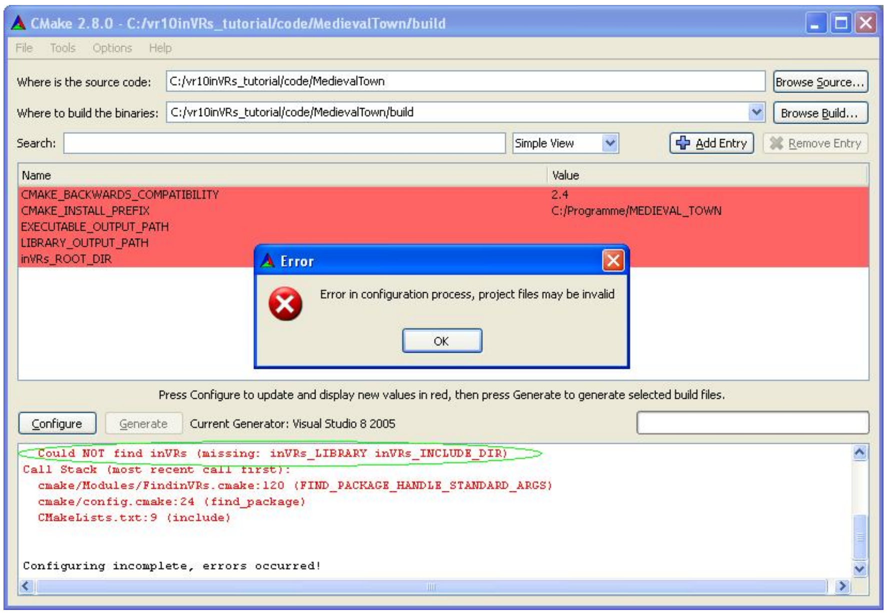
Specify native compilers

Specify toolchain file for cross-compiling

Specify options for cross-compiling

Figure 14.2: Select target project type

During the configuration CMake may stop printing an error message which says that the inVRs installation was not found on your system (green circle in the picture below):
If this happens you have to manually enter the path of your inVRs installation (note that inVRs must be installed in this directory, so when you build it from source also ensure that you build the **INSTALL** target of inVRs).
Afterwards press Configure again:
Finally if the configuration process was successfull you can generate your build system files by



Where is the source code: C:/vr10inVRs_tutorial/code/MedievalTown

Where to build the binaries: C:/vr10inVRs_tutorial/code/MedievalTown/build

Search:

Name	Value
CMAKE_BACKWARDS_COMPATIBILITY	2.4
CMAKE_INSTALL_PREFIX	C:/Programme/MEDIEVAL_TOWN
EXECUTABLE_OUTPUT_PATH	
LIBRARY_OUTPUT_PATH	
inVRs_ROOT_DIR	

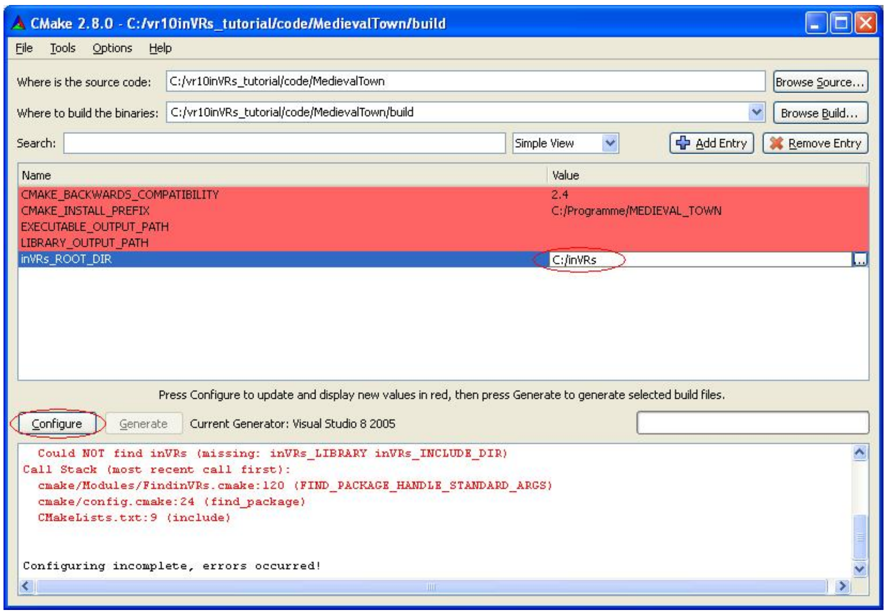
Error in configuration process, project files may be invalid

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Visual Studio 8 2005

```
  Could NOT find inVRs (missing: inVRs_LIBRARY inVRs_INCLUDE_DIR)
Call Stack (most recent call first):
  cmake/Modules/FindinVRs.cmake:120 (FIND_PACKAGE_HANDLE_STANDARD_ARGS)
  cmake/config.cmake:24 (find_package)
  CMakeLists.txt:9 (include)


Configuring incomplete, errors occurred!
```

Figure 14.3: Error message when inVRs installation was not found



Where is the source code: C:/vr10inVRs_tutorial/code/MedievalTown

Where to build the binaries: C:/vr10inVRs_tutorial/code/MedievalTown/build

Search:

Name	Value
CMAKE_BACKWARDS_COMPATIBILITY	2.4
CMAKE_INSTALL_PREFIX	C:/Programme/MEDIEVAL_TOWN
EXECUTABLE_OUTPUT_PATH	
LIBRARY_OUTPUT_PATH	
inVRs_ROOT_DIR	C:/inVRs

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Visual Studio 8 2005

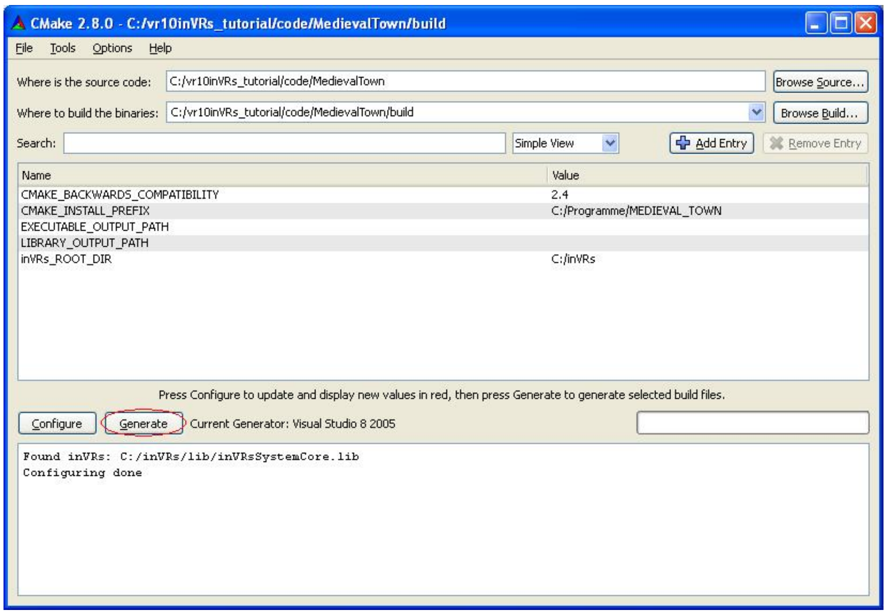
```
  Could NOT find inVRs (missing: inVRs_LIBRARY inVRs_INCLUDE_DIR)
Call Stack (most recent call first):
  cmake/Modules/FindinVRs.cmake:120 (FIND_PACKAGE_HANDLE_STANDARD_ARGS)
  cmake/config.cmake:24 (find_package)
  CMakeLists.txt:9 (include)


Configuring incomplete, errors occurred!
```

Figure 14.4: Finalize configuration

pressing the Generate button:
The generated build files (Makefiles, or Visual Studio projects) can be found in the subfolder *build*.

Windows: Open Tutorial Project in Visual Studio

Open Visual Studio and open the tutorial project, e.g. `<MedievalTown>/build/MedievalTown.sln`.
To create the application build the target **MedievalTown** or **ALL_BUILD**.



Where is the source code: C:/vr10inVRs_tutorial/code/MedievalTown

Where to build the binaries: C:/vr10inVRs_tutorial/code/MedievalTown/build

Search:

Simple View

Name	Value
CMAKE_BACKWARDS_COMPATIBILITY	2.4
CMAKE_INSTALL_PREFIX	C:/Programme/MEDIEVAL_TOWN
EXECUTABLE_OUTPUT_PATH	
LIBRARY_OUTPUT_PATH	
inVRs_ROOT_DIR	C:/inVRs

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Visual Studio 8 2005

```
Found inVRs: C:/inVRs/lib/inVRsSystemCore.lib
Configuring done
```

Figure 14.5: Generate build files

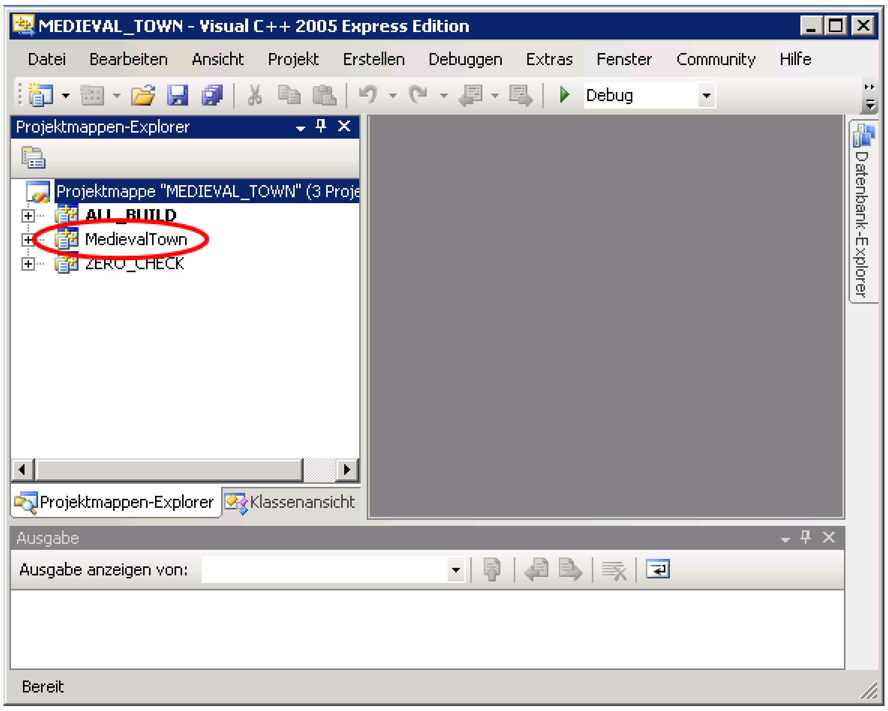


Figure 14.6: Build application

Linux, Mac OSX: Build sources using Makefiles

For building the tutorials using the generated Unix Makefiles enter the *build* subdirectory and call make:

```
# from within tutorial folder
cd build
```

```
make
```

Running the application

To start the application open the file `startTutorial.bat` on Windows or the file `startTutorial.sh` on Linux or Mac OSX in an editor. In the file you have to configure the path to your OpenSG and your inVRs installation. The paths are needed in the following lines in order to set the library paths needed by the application in order to find the dll-files:

```
SET OPENSG_DIR=C:\\Programme\\OpenSG
```

After you entered the correct paths you can start the application by executing the batch file.