–

Medieval Town Tutorial

Christoph Anthes, Roland Landertshamer,
and Marina Lenger

August 10, 2009

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.

Contents

Abstract **1**

Contents **1**

1 Introduction **1**
1.1 Tutorial Overview 1
1.2 Outline 3

2 Architecture Overview **5**
2.1 System Core 5
2.1.1 Databases 6
2.1.2 Communication 6
2.2 Interfaces 7
2.2.1 Input Interface 7
2.2.2 Output Interface 7
2.3 Modules 7
2.3.1 Navigation 8
2.3.2 Interaction 8
2.3.3 Network 8

3 Basic Application Development **9**
3.1 Using OpenSG and GLUT 10
3.2 Configuring inVRs 10
3.3 Working with the WorldDatabase 14
3.4 Summary 17

4 Navigation and Skybox **18**
4.1 Adding inVRs Components 18
4.2 Navigation 20
4.2.1 Managing User Input 23
4.3 Skybox 24
4.4 Summary 26

5 Transformation Management **27**
5.1 Height and Collision Maps 27
5.1.1 Generating Collision Maps 28
5.1.2 Generating Height Maps 29
5.2 Using Modifiers and Pipes 29
5.3 Summary 32

6 Interaction **33**
6.1 State Machine . . . 33
6.2 Implementing Interaction . . . 35
6.3 Events . . . 37
6.4 Summary . . . 38

7 Using Network Communication **39**
7.1 Concepts . . . 39
7.2 Setting up the Network Communication . . . 39
7.3 Transmitting Data . . . 40
7.4 Displaying Avatars . . . 41
7.5 Execution . . . 42
7.6 Summary . . . 42

8 Developing own Application Logic **43**
8.1 Input and Animations . . . 43
8.2 Summary . . . 44

9 Outlook **46**
9.1 Tools . . . 46
9.2 Going Immersive . . . 46
9.3 Acknowledgments . . . 47

Bibliography **48**

List of Figures **50**

Listings **51**

Appendix **53**

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.

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

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

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. An additional tutorial on how to enhance *inVRs* will be provided soon. 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

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.

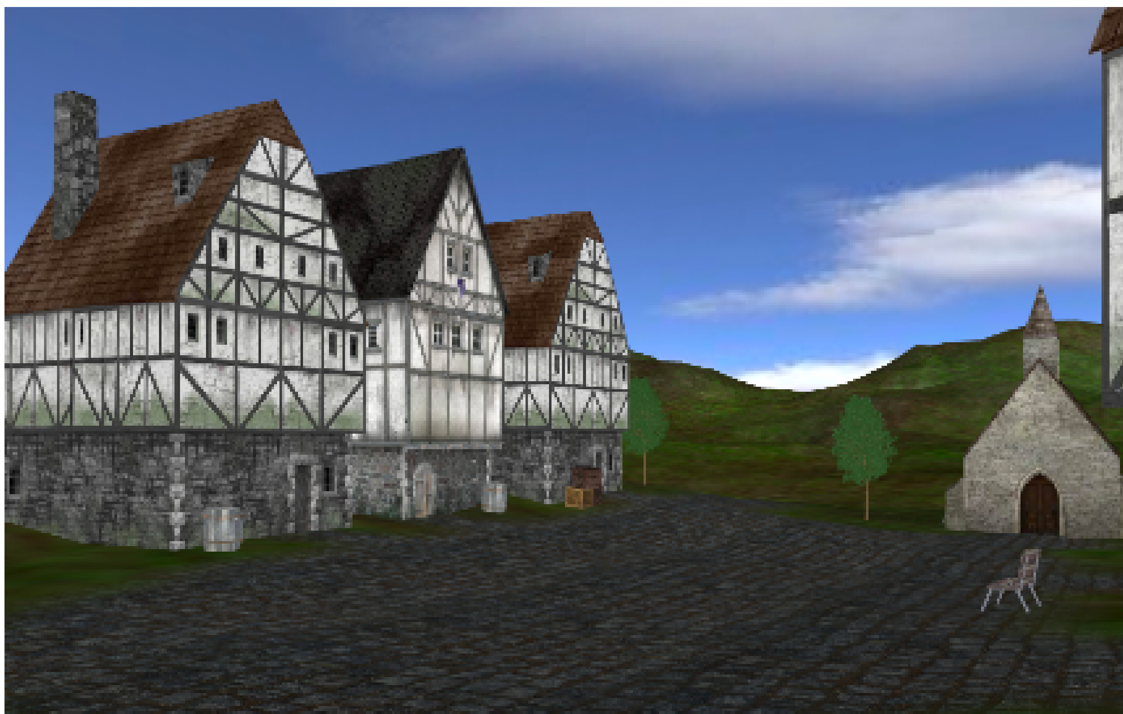

Figure 1.1: The Tutorial Town

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.
The following outline briefly describes the tutorial course.

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 individual 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 - Outlook
  The taught aspects of *inVRs* are recaptured and a brief outlook on what else could be explored in the framework is given. The additional tools 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.

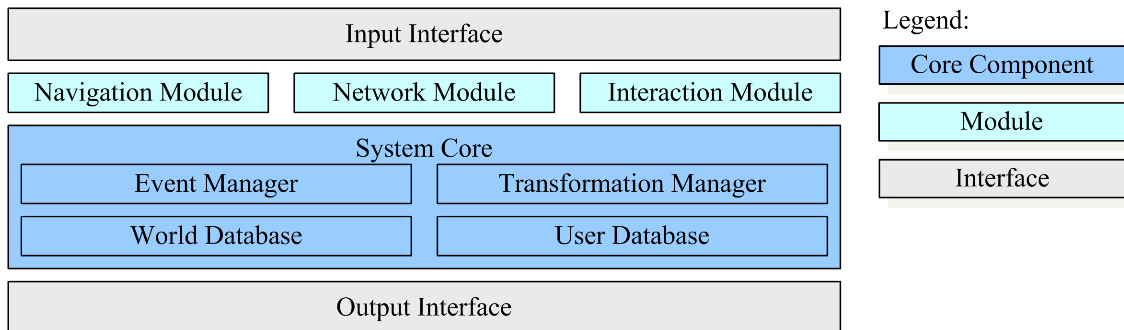

Figure 2.1: The Basic *inVRs* Components

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].

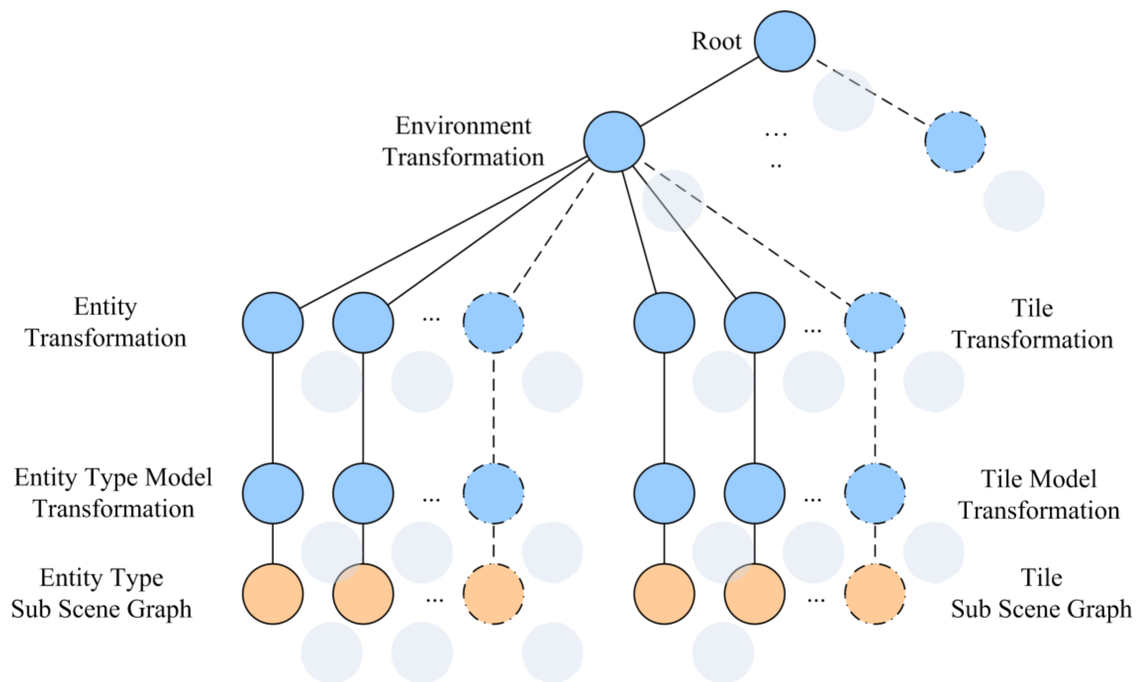

Figure 2.2: The Transformation Hierarchy of the World Database

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.

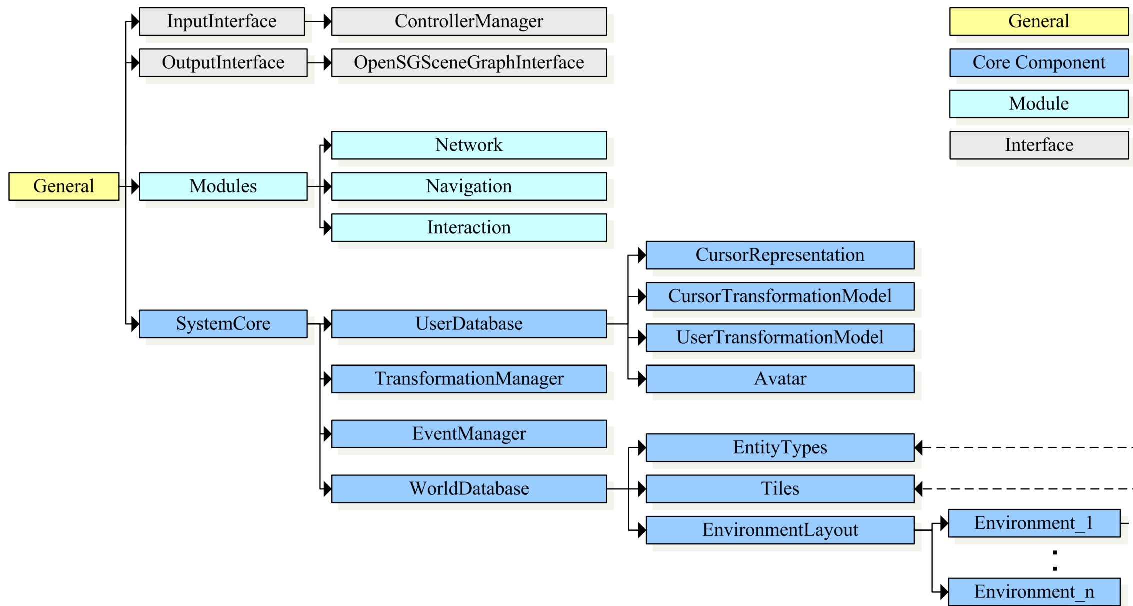

Figure 3.1: The *inVRs* Configuration Hierarchy

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

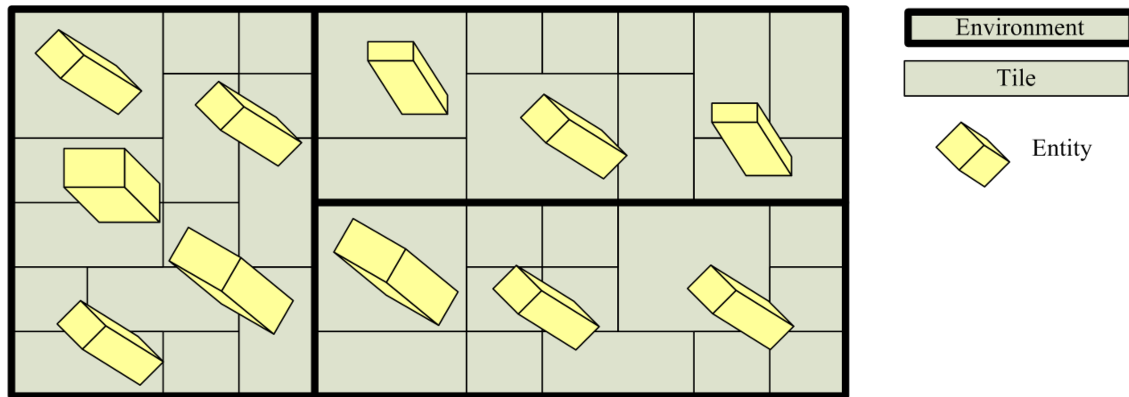

Figure 3.2: The World Database seen from Top

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

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 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.

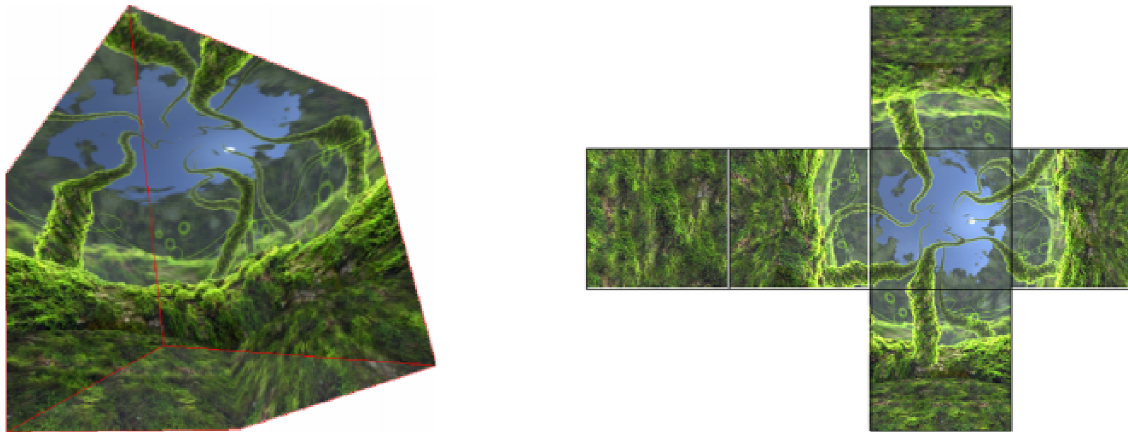

Figure 4.1: An Example Skybox

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. Doesn't that make you happy? Well, there is still much more to come.

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.

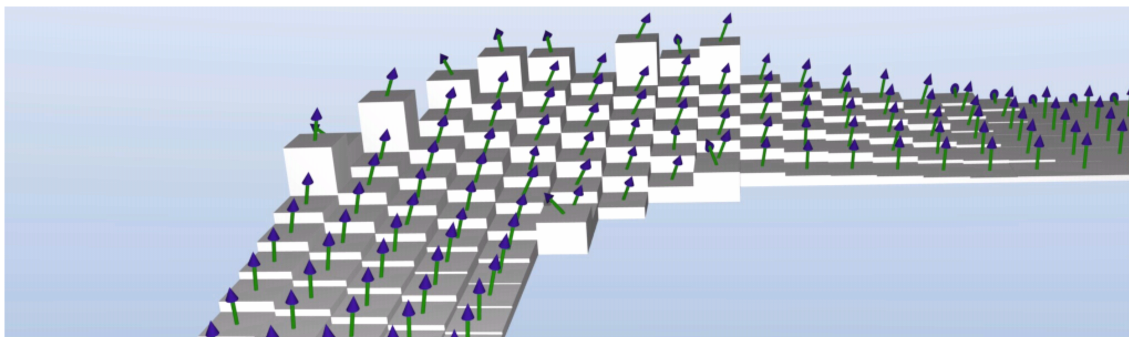

Figure 5.1: An Example Height Map

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.

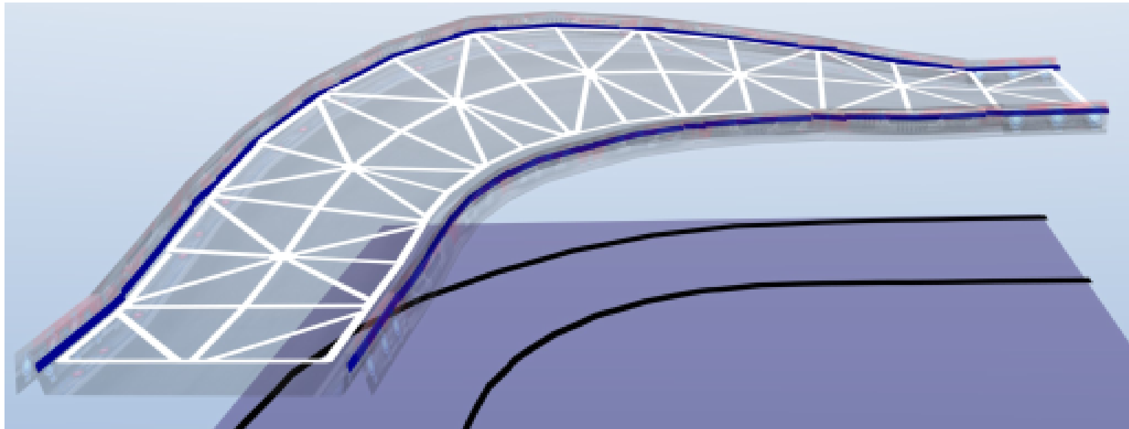

Figure 5.2: A Collision Map

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/

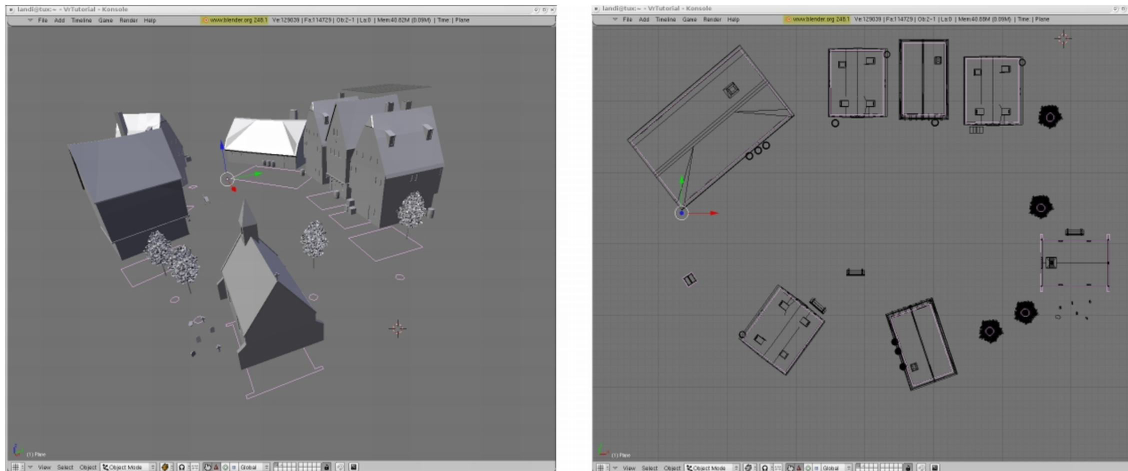

Figure 5.3: Generation of a Collision Map

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 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.

```
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.

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.

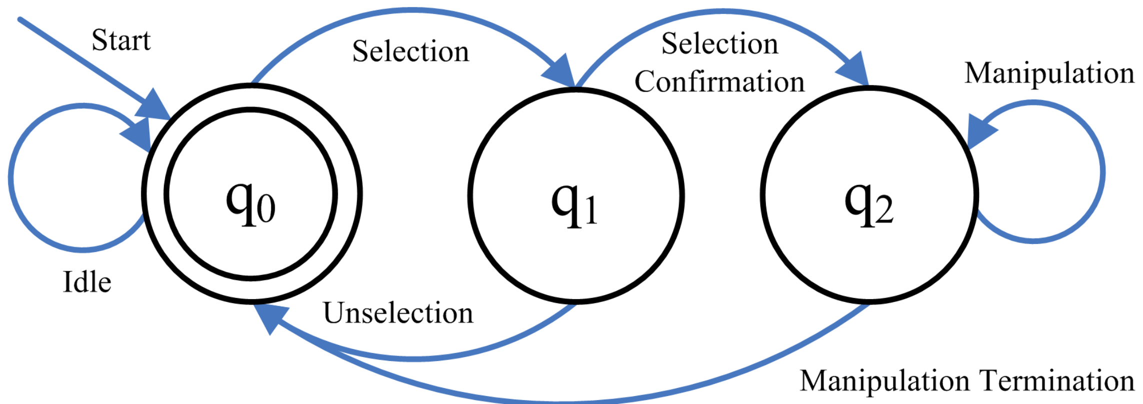

Figure 6.1: The Interaction State Machine

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->step(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. 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.

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 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.

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 windMillSpeed, 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. 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.

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.

[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]

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.

Chapter 9

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.
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 a huge variety of different modules and tools exist.

9.1 Tools

There are many. A large variety of tools exist which accompany the *inVRs* framework. These tools will be made available and documented in the very near future.
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], 3D menus as well as the introduced skybox, collision maps and height maps. And an extremely valuable tool for developing immersive multi-display applications is the CAVESceneManager [HJAA05] which acts as a wrapper for the OpenSG clustering functionality.
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.
The modules which have been developed so far cover 2D Physics, 3D Physics [ALV07], and animations.

9.2 Going Immersive

Well, the next step would be porting this tutorial on a stereoscopic [1] system with tracked input devices. You are way not as far away from this as you might think.

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

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.

9.3 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. 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.

[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.

[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.

[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.

List of Figures

1.1 The Tutorial Town 2

2.1 The Basic *inVRs* Components 5
2.2 The Transformation Hierarchy of the World Database 7

3.1 The *inVRs* Configuration Hierarchy 11
3.2 The World Database seen from Top 15

4.1 An Example Skybox 25

5.1 An Example Height Map 27
5.2 A Collision Map 28
5.3 Generation of a Collision Map 29

6.1 The Interaction State Machine 33

9.1 Open CMake and set MedievalTown path 55
9.2 Select target project type 56
9.3 Finalize configuration 56
9.4 Build application 57
9.5 Open CMake and set MedievalTown path 58
9.6 Select target project type 59
9.7 Finalize configuration 59

Listings

3.1 MedievalTown.cpp - Top Part of `main`() . . . 10
3.2 MedievalTown.cpp - Bottom Part of `main`() . . . 10
3.3 general.xml . . . 11
3.4 CodeFile1.cpp - Snippet-1-1 → MedievalTown.cpp . . . 12
3.5 CodeFile1.cpp - Snippet-1-2 → MedievalTown.cpp . . . 13
3.6 CodeFile1.cpp - Snippet-1-3 → MedievalTown.cpp . . . 13
3.7 CodeFile1.cpp - Snippet-1-4 → MedievalTown.cpp . . . 14
3.8 general.xml - Enter Path to inVRs Libraries . . . 14
3.9 worldDatabase.xml . . . 15
3.10 entities.xml . . . 15
3.11 tiles.xml . . . 16
3.12 environmentLayout.xml . . . 16
3.13 environment.xml . . . 17
4.1 CodeFile2.cpp - Snippet-2-1 → MedievalTown.cpp . . . 18
4.2 Snippets2.xml - Snippet-2-1 → general.xml . . . 19
4.3 Snippets2.xml - Snippet-2-2 → general.xml . . . 19
4.4 Snippets2.xml - Snippet-2-3 → general.xml . . . 19
4.5 CodeFile2.cpp - Snippet-2-2 → MedievalTown.cpp . . . 19
4.6 CodeFile2.cpp - Snippet-2-3 → MedievalTown.cpp . . . 20
4.7 CodeFile2.cpp - Snippet-2-4 → MedievalTown.cpp . . . 20
4.8 Snippets2.xml - Snippet2-4 → userDatabase.xml . . . 21
4.9 Snippets2.xml - Snippet2-5 → modules.xml . . . 21
4.10 CodeFile2.cpp - Snippet-2-5 → MedievalTown.cpp . . . 21
4.11 CodeFile2.cpp - Snippet-2-6 → MedievalTown.cpp . . . 22
4.12 CodeFile2.cpp - Snippet-2-7 → MedievalTown.cpp . . . 22
4.13 CodeFile2.cpp - Snippet-2-8 → MedievalTown.cpp . . . 22
4.14 CodeFile2.cpp - Snippet-2-9 → MedievalTown.cpp . . . 23
4.15 CodeFile2.cpp - Snippet-2-10 → MedievalTown.cpp . . . 23
4.16 CodeFile2.cpp - Snippet-2-11 → MedievalTown.cpp . . . 23
4.17 CodeFile2.cpp - Snippet-2-12 → MedievalTown.cpp . . . 23
4.18 CodeFile2.cpp - Snippet-2-13 → MedievalTown.cpp . . . 23
4.19 CodeFile2.cpp - Snippet-2-14 → MedievalTown.cpp . . . 24
4.20 navigation.xml . . . 24
4.21 CodeFile2.cpp - Snippet-2-15 → MedievalTown.cpp . . . 25
4.22 CodeFile2.cpp - Snippet-2-16 → MedievalTown.cpp . . . 25
4.23 CodeFile2.cpp - Snippet-2-17 → MedievalTown.cpp . . . 25
5.1 CodeFile3.cpp - Snippet-3-1 → MedievalTown.cpp . . . 29
5.2 modifiers.xml . . . 30
5.3 Snippets3.xml - Snippet-3-1 → modifiers.xml . . . 31
5.4 Snippets3.xml - Snippet-3-2 → modifiers.xml . . . 31
5.5 Snippets3.xml - Snippet-3-3 → modifiers.xml . . . 31
5.6 CodeFile3.cpp - Snippet-3-2 → MedievalTown.cpp . . . 31

5.7 CodeFile3.cpp - Snippet-3-3 → MedievalTown.cpp 32
6.1 interaction.xml 34
6.2 Snippets4.xml - Snippet-4-1 → general.xml 35
6.3 Snippets4.xml - Snippet-4-2 → modules.xml 35
6.4 CodeFile4.cpp - Snippet-4-1 → MedievalTown.cpp 35
6.5 Snippets4.xml - Snippet-4-3 → userDatabase.xml 36
6.6 homerCursorModel.xml 36
6.7 Snippets4.xml - Snippet-4-4 → modifiers.xml 36
6.8 Snippets4.xml - Snippet-4-5 → modifiers.xml 37
6.9 Snippets4.xml - Snippet-4-6 → general.xml 37
6.10 CodeFile4.cpp - Snippet-4-2 → MedievalTown.cpp 37
7.1 Snippets5.xml - Snippet-5-1 → modules.xml 39
7.2 Snippets5.xml - Snippet-5-2 → modules.xml 39
7.3 CodeFile5.cpp - Snippet-5-1 → MedievalTown.cpp 40
7.4 CodeFile5.cpp - Snippet-5-2 → MedievalTown.cpp 40
7.5 CodeFile5.cpp - Snippet-5-3 → MedievalTown.cpp 40
7.6 network.xml 40
7.7 Snippets5.xml - Snippet-5-3 → modifiers.xml 41
7.8 Snippets5.xml - Snippet-5-4 → modifiers.xml 41
7.9 Snippets5.xml - Snippet-5-5 → modifiers.xml 41
7.10 avatar.xml 42
8.1 CodeFile6.cpp - Snippet-6-1 - Top Part → MedievalTown.cpp 43
8.2 CodeFile6.cpp - Snippet-6-1 - Bottom Part → MedievalTown.cpp 44

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: Linux

Prerequisites

To be able to build the tutorial application inVRs has to be built and installed. For details please have a look at the inVRs installation instructions.

Step 1 (OPTIONAL): Configure Application

Before building the Tutorial application the path to the inVRs installation has to be configured. If you are building the MedievalTown Tutorial within the `inVRs/tutorials` folder you can skip this step and continue with **Step 2**.
Otherwise please open the file `user.cmake` and define the path to the inVRs installation:

```
# 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_DIR /usr/local/src/inVRs)
```

Step 2: Create Makefiles using CMake

Before the application can be built CMake has to be used in order to create the Makefiles. Therefore enter the `build` directory and call cmake:

```
cd build
cmake ../
```

This will create the needed Makefiles for building the MedievalTown application in the `build` folder.

Step 3: Build Application

After the Makefiles are created you can build the application by calling the `make` command in the `build` directory:

```
cd build
make
```

Step 4: Start Application

For starting the application first open the file `startTutorial.sh`. In this file please configure the path where the OpenSG libraries can be found, e.g.:

```
OPENSG_LIB_PATH=/usr/local/lib/opt
```

If you are not building the MedievalTown Tutorial inside the `inVRs/tutorials` folder you will also have to modify the path to your *inVRs* installation in this file.
Afterwards you can start the application by executing the `startTutorial.sh` script.

Installation Instructions: Windows

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`

Prerequisites

To be able to build the tutorial application inVRs has to be built and installed. For details please have a look at the inVRs installation instructions.

Step 1 (OPTIONAL): Configure Application

Before building the Tutorial application the path to the inVRs installation has to be configured. If you are building the MedievalTown Tutorial within the `inVRs/tutorials` folder you can skip this step and continue with **Step 2**.
Otherwise please open the file `user.cmake` and define the path to the inVRs installation:

```
# 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_DIR C:/Program\ Files/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.

Step 2: Create Visual Studio Project files with CMake

Open CMake and set the paths to the MedievalTown project:

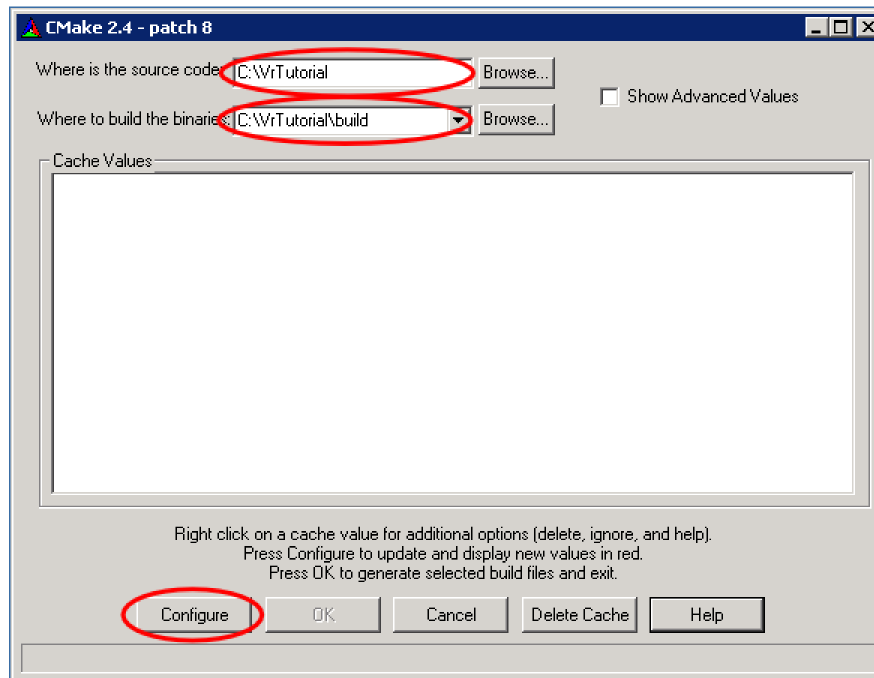

Figure 9.1: Open CMake and set MedievalTown path

Press Configure and select as target build type Microsoft Visual Studio 2005.

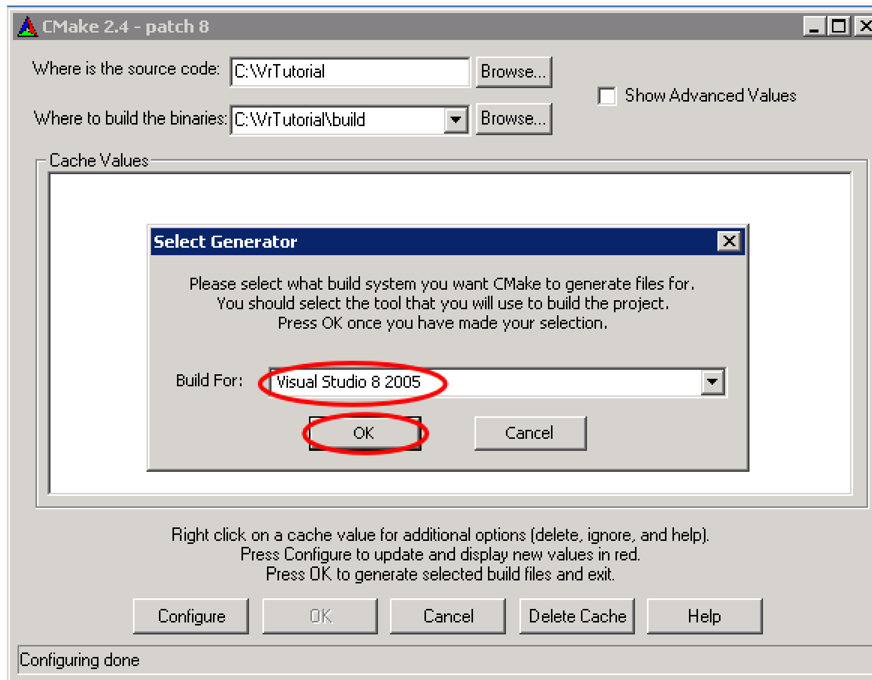

Figure 9.2: Select target project type

If an error occurs finding inVRs please ensure that the path entered in step 1 is set correctly. Also ensure that inVRs is built (target **ALL_BUILD**) and installed (target **INSTALL**) already. In case Configure finished without errors press Configure again and afterwards press OK.

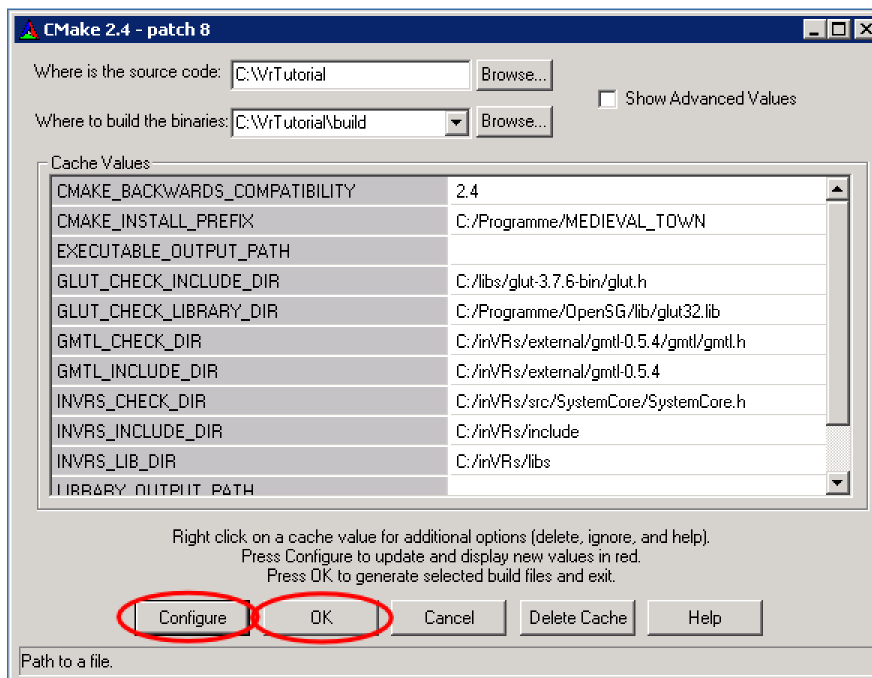

Figure 9.3: Finalize configuration

Step 3: Open MedievalTown Project in Visual Studio

Open Visual Studio and open the VrTutorial project from ⟨`MedievalTown`⟩`/build/MedievalTown.sln`.

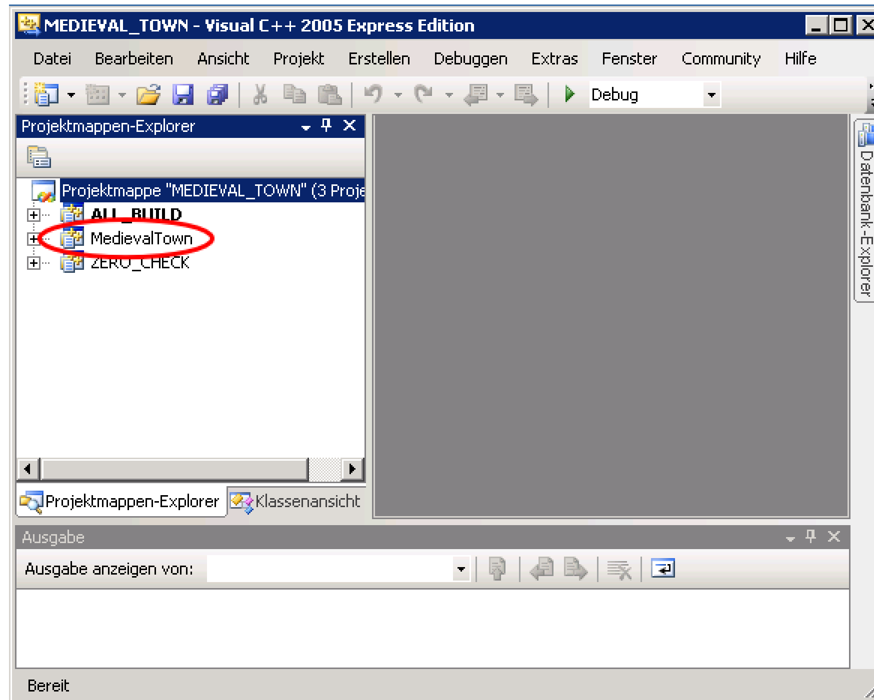

Figure 9.4: Build application

To create the application build the target **MedievalTown**. For building the fully implemented tutorial you can also build the target **MedievalTownFinal**.

Step 4: Running the application

To start the application open the file `startTutorial.bat` in an editor. In the file you have to configure the path to your OpenSG installation. If you are not building the MedievalTown application within the `inVRs/tutorials` folder you will also have to configure the path to 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.

Installation Instructions: Mac OS X

Prerequisites

To be able to build the tutorial application inVRs has to be built and installed. For details please have a look at the inVRs installation instructions.

Step 1 (OPTIONAL): Configure Application

Before building the Tutorial application the path to the inVRs installation has to be configured. If you are building the MedievalTown Tutorial within the `inVRs/tutorials` folder you can skip this step and continue with **Step 2**.
Otherwise please open the file `user.cmake` and define the path to the inVRs installation:

```
# 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_DIR /Users/guest/inVRs)
```

Step 2: Create Makefiles with CMake

Open CMake and set the paths to the MedievalTown project:

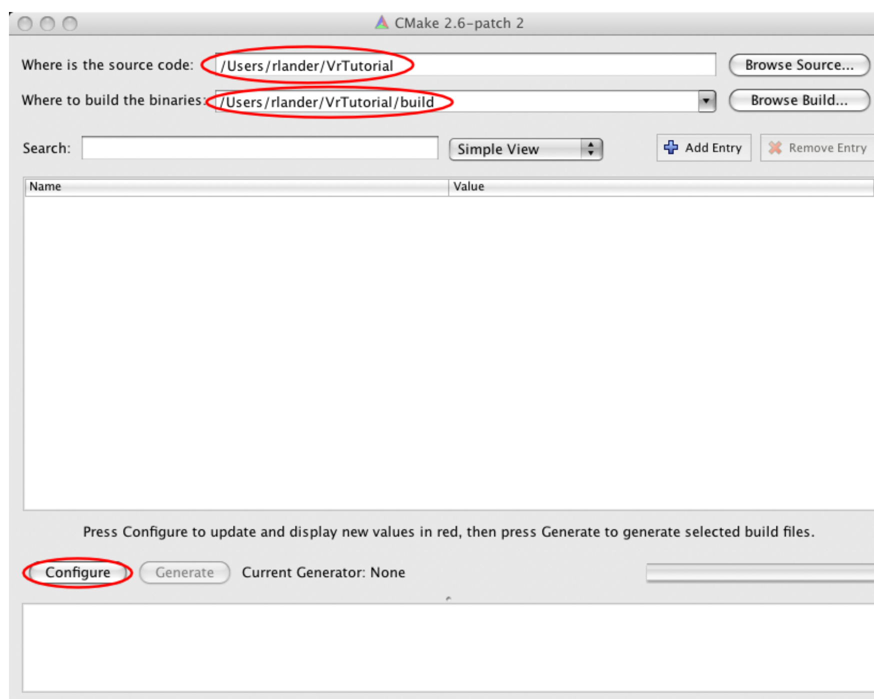

Figure 9.5: Open CMake and set MedievalTown path

Press Configure and select as target build type Unix Makefiles.

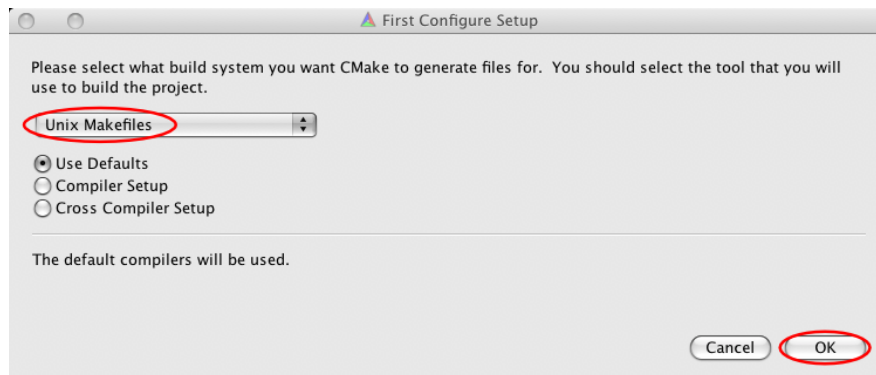

Figure 9.6: Select target project type

If an error occurs finding inVRs please ensure that the path entered in step 1 is set correctly. Also ensure that inVRs is built (target **ALL_BUILD**) and installed (target **INSTALL**) already. In case Configure finished without errors press Configure again and afterwards press Generate.

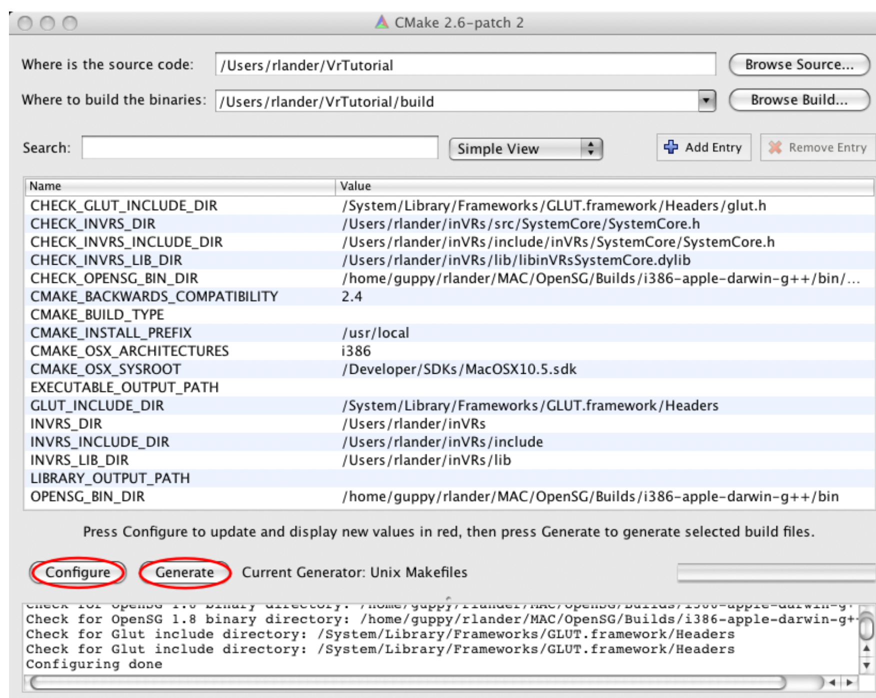

Figure 9.7: Finalize configuration

Step 3: Build Application

After the Makefiles are created you can build the application by calling the `make` command in the `build` directory:

```
cd build
make
```

Step 4: Start Application

For starting the application first open the file `startTutorial.sh`. In this file please configure the path where the OpenSG libraries can be found, e.g.:

```
OPENSG_LIB_PATH=/usr/local/lib/opt
```

If you are not building the MedievalTown Tutorial inside the `inVRs/tutorials` folder you will also have to modify the path to your *inVRs* installation in this file.
Afterwards you can start the application by executing the `startTutorial.sh` script.