—

Configuring the inVRs Framework

Christoph Anthes and Roland Landertshamer

September 14, 2009

Abstract

The *inVRs* framework is a complex software framework with complex configuration architecture. To ease the creation of *inVRs* applications this document provides a deep inside in the configuration mechanisms. It should be used as a manual for configuring own *inVRs* applications.
The manual introduces all XML based configuration files of the basic *inVRs* components. Additional modules or tools might have to be configured in a different manner. Details on these additional components can be found in their specific manuals.

Contents

Chapter 1

Introduction

Although the *inVRs* [AV06] framework is able to take away a fair bit of coding from the application developer it requires initially a significant amount of configuration effort. An ordinary *inVRs* application makes use of roughly 20 to 25 configuration files of which most are fortunately rather generic. This document describes the setup of the individual framework components.
The configuration of the framework is performed in an eXtensible Markup Language (XML) format, which is described in detail in this manual. The according Document Type Definition (DTD) files for checking the correctness of configuration can be found in the Appendix or online under: `http://dtd.invrs.org`. These DTD files should be used for verification of the *inVRs* configuration. Verification of the configuration files should be done externally before application execution.

1.1 Configuration Overview

The hierarchy which is typically used for the configuration files of the *inVRs* framework is illustrated in Figure 1.1. It is not to be used only on a file basis. In general it is recommended to have a separate sub-directory for each of the file types illustrated as boxes in the diagram.

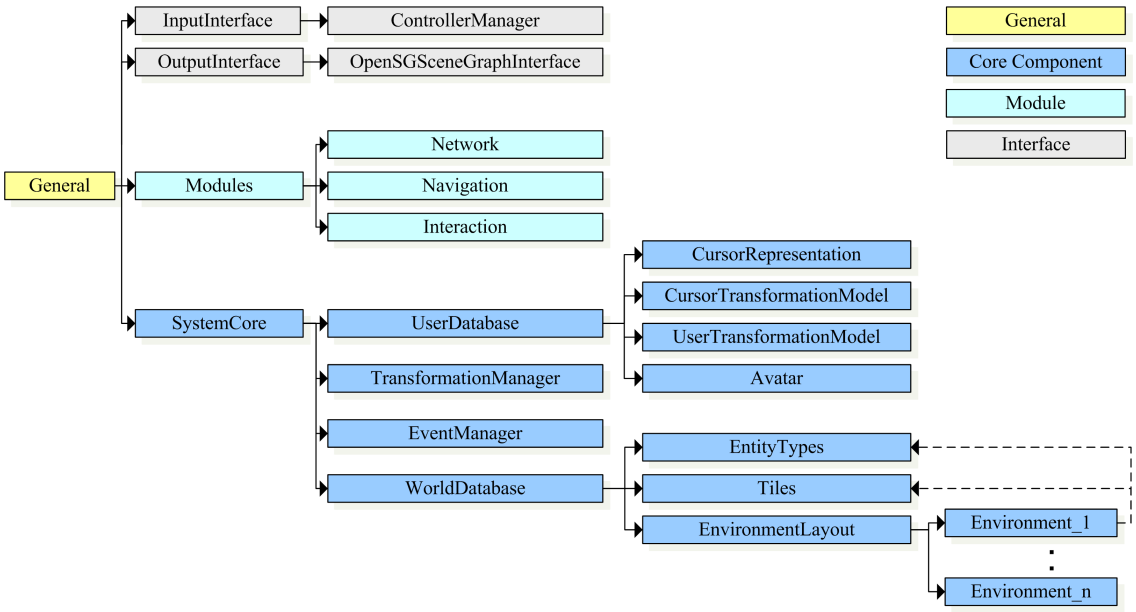

Figure 1.1: The *inVRs* Configuration Hierarchy

It is often desirable to have different configurations which can be easily exchanged stored in the same directory. Common use cases are the exchange of the whole navigation or interaction setup, the controllers or displays. Four main categories of configuration files exist:

- General
  The general configuration as illustrated in yellow, takes care of the path setting for the configuration as well as the content used by *inVRs* applications. It provides links to the other used *inVRs* components.
- Interfaces
  The interfaces, drawn in grey, are subdivided into the input interface which takes care of the input devices used to control the applications and the output interface handling the output devices which are used to display the virtual environment.
- Modules
  The modules, shown in light blue, consist by default of the navigation module, the interaction module and the network module. Additional modules as for example for physics simulation would be configured here as well.
- Core components
  The core components, colored in dark blue, are used for data storage and message handling inside the framework. The user database and the world database store information about the virtual world, while the transformation manager and the event manager take care of distribution between the interfaces, modules and the other core components.

Additional user-defined configurations which are dedicated to an application should be stored in a separate directory.

1.2 Concepts and Considerations

Some basic concepts used for the configuration of the *inVRs* framework will be explained before we take a close look at the specific configuration files.

1.2.1 Argument Vector

The argument vector, implemented in the class `ArgumentVector`, describes an undefined amount of attributes belonging to the parent XML node. It provides a list of triples where the first attribute `key` indicates the name of the argument, the second attribute identified by `type` is the datatype of the argument and the last attribute of the <argument>-node `value` provides the actual value for the argument specified in `key`, which has to be of the previously defined datatype.
Argument vectors are often used for the configuration of exchangeable components like navigation models and interaction models.

```
<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>
```

Listing 1.1: Example of an Argument Vector

1.2.2 Transformation Node

To perform geometrical transformations a transformation node is used in the *inVRs* configuration. This node provides sub-nodes for translation, rotation, scale and scale orientation. It is implemented in the class `TransformationData`.

```
<transformation>
  <translation x="0" y="0" z="0"/>
  <rotation x="1" y="0" z="0" angleDeg="-90"/>
  <scale x="-0.08" y="-0.08" z="0.08"/>
</transformation>
```

Listing 1.2: Example for a Transformation Node

Each of the sub-nodes contains float attributes for x, y and z. The <rotation>-node configures a quaternion based rotation using an additional angle. This angle can either be given by the attribute `angleDeg` or alternatively `angleRad` determining whether the angle is defined in degrees or radiant.

1.2.3 Representation Node

Another common node is the <representation>-node, which contains data about a model and an additional transformation which are to be associated with the parent node. The node can contain the optional attribute `copy` describing whether the model can be copied[1] or whether just a reference to the model is given. It is implemented in the `ModelInterface` of the output interface.
The <file>-sub-node consists of two attributes. The first attribute `type` contains information about the file type of the 3D model which is to be associated, while the second attribute `name` identifies the filename of the model.

```
<representation>
  <file type="VRML97" name="tie.wrl" />
  <transformation>
    <translation x="0" y="0" z="0" />
    <rotation x="0" y="1" z="0" angleDeg="0" />
    <scale x="1" y="1" z="1" />
  </transformation>
</representation>
```

Listing 1.3: Example for a Representation Node

1.2.4 Naming

Some naming conventions are to be kept for configuring the framework. Typically the component or sub-component of the framework has the same name as the main node of its according configuration file. The name of the node should be identical or similar to the class name of the implementation of the component.
All node names are to begin with lower case letter. The names of the attributes are always written lower-case. More details on the naming aspect are given in the *Extending the inVRs Framework Manual*.

1.2.5 File References

In theory it would be possible to structure the configuration of the framework in a single file or at least fewer files than the ones which are used right now. When using such an implementation

[1] when OpenSG is used as a scene graph this operation is implemented as a deep clone

minor changes could cause significant rewriting of the file. For example if a navigation technique, which is used in an application should be exchanged completely the whole model setup would have to be exchanged. Thus the components often make use of references to other files.
In this case it is possible to change parts of the configuration files, which makes the application configuration extremely flexible and reusable. The users are able to create repositories of their configurations and expand them over the time.
On the other hand it is hard for inexperienced users to keep an overview on the configuration without the help of this document.

1.2.6 Versioning

Each configuration file or DTD carries a version of the current *inVRs* release. From the version 1.0 alpha4 on it is possible to upgrade configuration files automatically to the current *inVRs* version. This update mechanism has to be manually triggered in the general configuration which will be explained in the following chapter.
To identify the correct version each top node of the *inVRs* configuration files contains an attribute `version` which keeps the current version number.

1.3 Outline

The chapters of this document cover the following topics:

- Chapter 2 - System Core
  The configuration of the system core with its sub-components the world database the user database and the transformation manager is described in this chapter. The structure of the user database and the world database lead to a variety of sub-configurations which are described as well.
  To configure the transformation manager a variety of possibilities exist. These configuration components are described in detail as well.

- Chapter 3 - Input Interface
  Arbitrary devices can be interconnected to *inVRs*. The data gathered by these devices is stored inside an abstract controller which is described in this chapter. All supported input libraries with their specifics are briefly introduced.

- Chapter 4 - Output Interface
  In order to render graphics or audio output *inVRs* makes use of an output interface. The configuration of this interface is provided in this chapter.

- Chapter 5 - Navigation Module
  The navigation module implements navigation by composing three different models for direction, orientation, and speed into a resulting navigation technique. The configuration of the available *inVRs* standard models including their parameterization is described in this chapter.

- Chapter 6 - Interaction Module
  The interaction module of *inVRs* is implemented as an automaton. The configuration of this automaton is given by setting up state transition functions in order to implement interaction techniques.

- Chapter 7 - Network Module
  The configuration of the standard network module of the *inVRs* framework is described briefly in this chapter.

- Chapter 8 - Outlook
  The final chapter outlook recaptures the main aspects of this document and offers suggestions for the creation of own application specific configurations as well as the extensions of the already provided configurations.

Chapter 2

System Core and General Setup

The first configuration file you probably want to modify for your specific setup of an application is the general configuration which provides a basic path setup as well as the links to the configurations of the *inVRs* specific components as well as the user defined components. It is parsed and evaluated by the `Configuration` class of the `SystemCore`.
The two main nodes are:

- <general>
  The <general>-node can contain up to six sub-nodes which provide the filenames of the configuration files of the component types which are to be registered during the startup of an *inVRs* application. Additionally the configuration for the application base, which is wrapping setup functionality, and configuration upgrading is stored in this node.
  The sub-nodes of the <general>-node always contain <option>-sub-nodes which have attributes consisting of the couples of `key` and `value`. The attribute `key` describes the name of the attribute while `value` defines its contents.

  - <applicationBase>
    In the <option>-node with the key `profilerLogFile` of the application base the name of a log file used for profiling an *inVRs* application can be specified. The log file is stored in the directory where your *inVRs* application is executed. This file contains information about the run-time of several components. More detail will be provided in the *Extending the inVRs Framework* manual.
  - <openSGApplicationBase>
    In case OpenSG [Rei02] is used as a scene graph this node should be used. When the key `useCSM` is set to true the CAVESceneManager will be used. It acts as a wrapper for OpenSG multi-display support. More detail on this tool is provided in the *Going Immersive Tutorial*. The name of the configuration file of the CAVESceneManager [HJAA05] is specified in the value attribute of the according `csmConfigFile` key.
  - <interfaces>
    The <interfaces>-node contains typically two different <option>-sub-nodes which are identified by the two key attributes with the values `inputInterfaceConfiguration` and `outputInterfaceConfiguration`. These two keys take the names of the configuration files of the according interfaces as an attribute.
  - <modules>
    The <modules>-node contains one sub-node which can be identified by the attribute `modulesConfiguration`. This attribute should contain the name of the configuration file for the modules as value.
  - <systemCore>
    The <systemCore>-node contains a single sub-node which is identified by the key

`systemCoreConfiguration`. The name of the system core configuration file is stored in the according value.

- <XmlConfigLoader>
  In order to keep downward compatibility the XML parser of the *inVRs* framework allows to update deprecated configuration files to current versions. If the key `updateFiles` is set to true and a newer configuration version is available the configuration files that were parsed will be updated. Watch out when using this option. Currently *inVRs* does not transcribe the comments from old configuration files to the updated files.

- <paths>
  The <paths>-node contains the whole path setup for the *inVRs* framework in its sub-nodes. It can contain again two different types of sub-nodes.

  - <root>
    The <root>-node describes where the root path for the *inVRs* application is to be set. Most paths that are used are considered to be relative to the *inVRs* root path.
  - <path>
    Different categories of paths are common; they are all set relative to the root path. Typically the following categories of paths exist: world database, user database, transformation manager, interfaces, modules and models. Each <path>-node takes two attributes, `name` describing the name of the path and `directory` providing the actual path. More detail on this concept was provided in the introductory chapter.

Most *inVRs* applications make use of the application base concept as described in detail in the *Going Immersive Tutorial*. Additionally all three component types, the interfaces, modules and the core are used in an application. The following example shows a typical *inVRs* configuration file for `general.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generalConfig SYSTEM "http://dtd.invrs.org/generalConfig_v1.0a4.dtd">
<generalConfig version="1.0a4">
  <!-- This is the configuration for the inVRs Framework -->
  <general>
    <ApplicationBase>
      <option key="profilerLogFile" value="profile.log" />
    </ApplicationBase>
    <OpenSGApplicationBase>
      <option key="useCSM" value="true"/>
      <option key="csmConfigFile" value="cave.config"/>
    </OpenSGApplicationBase>
    <Interfaces>
      <option key="inputInterfaceConfiguration" value="inputInterface.xml" />
      <option key="outputInterfaceConfiguration" value="outputInterface.xml" />
    </Interfaces>
    <Modules>
      <option key="modulesConfiguration" value="modules.xml" />
    </Modules>
    <SystemCore>
      <option key="systemCoreConfiguration" value="systemCore.xml"/>
    </SystemCore>
    <XmlConfigLoader>
      <option key="updateFiles" value="true"/>
    </XmlConfigLoader>
  </general>

  <paths>
    <root directory="./" />
    <path name="SystemCoreConfiguration" directory="final/config/systemcore/" />
    <path name="Plugins" directory="../../lib"/>
    <path name="ModulesConfiguration" directory="final/config/modules/" />
    <path name="InputInterfaceConfiguration"
```

```
      directory="final/config/inputinterface" />
    <path name="OutputInterfaceConfiguration"
      directory="final/config/outputinterface" />
    <!-- Paths for World DB Datastructure-->
    <path name="WorldConfiguration"
      directory="final/config/systemcore/worlddatabase/" />
    <path name="EnvironmentConfiguration"
      directory="final/config/systemcore/worlddatabase/environment/" />
    <path name="EntityTypeConfiguration"
      directory="final/config/systemcore/worlddatabase/entityTypes/" />
    <path name="TileConfiguration"
      directory="final/config/systemcore/worlddatabase/tile/" />
    <!-- Paths for User DB Datastructure-->
    <path name="UserConfiguration"
      directory="final/config/systemcore/userdatabase/" />
    <path name="AvatarConfiguration"
      directory="final/config/systemcore/userdatabase/avatar/" />
    <path name="CursorConfig"
      directory="final/config/systemcore/userdatabase/cursorRepresentation/" />
    <path name="CursorTransformationModelConfiguration"
      directory="final/config/systemcore/userdatabase/cursorTransformationModel/" /
          >
    <path name="UserTransformationModelConfiguration"
      directory="final/config/systemcore/userdatabase/userTransformationModel/" />
    <!-- Path for TransformationManager -->
    <path name="TransformationManagerConfiguration"
      directory="final/config/systemcore/transformationmanager/" />
    <!-- Path for Interfaces Datastructure -->
    <path name="ControllerConfiguration"
      directory="final/config/inputinterface/controllermanager/" />
    <!-- Paths for Module Datastructure -->
    <path name="NavigationConfiguration"
      directory="final/config/modules/navigation/" />
    <path name="InteractionModuleConfiguration"
      directory="final/config/modules/interaction/" />
    <path name="NetworkConfiguration" directory="final/config/modules/network/" />
    <!-- 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="Images" directory="images/" />
  </paths>
</generalConfig>
```

Listing 2.1: general.xml

2.1 Modules

Let's have brief look at the structure of the general module configuration, before we continue with the system core. The modules configuration as referenced by the <modules>-node with the `key` attribute set to `modules` in the general configuration contains links to the configuration files of the individual modules. Each of the modules is identified by a <module>-node and can take up to three parameters:

- `name`
  This attribute describes the name of the module. In the standard *inVRs* package it can be set either to `Navigation`, `Interaction`, or `Network`.

- `configFile`
  The name of the configuration file of the specific module. These files will be explained in the subsequent chapters dedicated to the according modules.

- `libraryName`
  In case the name of the library is different that the module name, this attribute allows to specify the library name for the module to be loaded. These libraries should be located in the directory indicated in the general configuration file which has the <path>-nodes `name` attribute set to `Plugins`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE modules SYSTEM "http://dtd.invrs.org/modules_v1.0a4.dtd">
<modules version="1.0a4">
  <module name="Navigation" configFile="navigation.xml"/>
  <module name="Interaction" configFile="interaction.xml" />
</modules>
```

Listing 2.2: exampleModules.xml

2.2 System Core

The system core configuration is used for referencing to the configuration files of the individual core components of the *inVRs* framework. It is consequentially parsed by the cores main class `SystemCore`. The directory of the configuration of the system core can be specified in the <path>-sub-node with the name attribute `SystemCoreConfiguration` in the general configuration.

- <worldDatabase>
  This node takes one `configFile` attribute. Its value describes the filename of the configuration of the world database. This configuration file is parsed and evaluated by the class `WorldDatabase`.

- <userDatabase>
  This node takes one `configFile` attribute, which is the filename of the configuration of the user database. This configuration file is parsed and evaluated by the class `UserDatabase`.

- <transformationManager>
  This node takes one `configFile` attribute, which is the filename pointing to the transformation manager's configuration. This configuration file is then parsed and evaluated by the class `TransformationManager`.

```
<?xml version="1.0"?>
<!DOCTYPE systemCore SYSTEM "http://dtd.invrs.org/systemCore_v1.0a4.dtd">
<systemCore version="1.0a4">
  <worldDatabase configFile="worldDatabase.xml" />
  <userDatabase configFile="userDatabase.xml" />
  <transformationManager configFile="modifiers.xml" />
</systemCore>
```

Listing 2.3: systemCore.xml

2.3 User Database

The user database configuration is used for referencing to the configuration of the individual parts of the user database of the *inVRs* framework. The configuration of the user database is parsed and evaluated in an object of the class `UserDatabase`. In case a path for the user database configuration is set as one of the <path>-sub-nodes in the general configuration this directory is searched for the configuration file specified in the <userDatabase>-node of the system core configuration. It provides four nodes containing references to configuration files of the user representation, the cursor representation, the model for defining the behavior of the cursor, and the model describing the transformation of the user:

- <avatar>
  Two different types of avatars are supported so far by the framework, simple avatars and *Avatara* avatars. This node contains the file name of the avatar configuration. Depending on the chosen configuration in the referenced file a given type of avatar is loaded and used for the rendering of the user representation.

- <cursorRepresentation>
  The cursor representation allows the user to chose a way how the own cursor is represented. Two alternatives are offered so far. A simple cursor as well as a cursor representing the state of interaction can be used. These representations are loaded as well dynamically based on the referenced configuration file.

- <cursorTransformationModel>
  A variety of cursor transformation models are available for *inVRs*. These cursor models describe the behavior of the cursor during interaction and navigation through the scene. These models are often very closely related to the interaction technique which is used.

- <userTransformationModel>
  To work with the *inVRs* coordinate systems of the user the user transformation model is often used. It describes the position and orientation of the user mapped inside the VE.

The following listing gives an example for the configuration of the user database:

```
<?xml version="1.0"?>
<!DOCTYPE userDatabase SYSTEM "http://dtd.invrs.org/userDatabase_v1.0a4.dtd">
<userDatabase version="1.0a4">
  <avatar configFile="undead.xml" />
  <cursorRepresentation configFile="interactionCursor.xml" />
  <cursorTransformationModel configFile="virtualHandCursorModel.xml"/>
  <userTransformationModel configFile="headPositionUserTransformationModel.xml"/>
</userDatabase>
```

Listing 2.4: exampleUserDatabase.xml

The following subsections will describe in detail the configurations of the individual components used in the user database.

2.3.1 Avatar

The configuration of the avatar is passed down from the class `AvatarInterface` to the derived implementations.
Depending on the chosen type of the main node two different alternatives are possible:

- SimpleAvatar
  The simple avatar displays a basic geometry which is used as a representation for the user. The configuration is parsed by objects of the class `SimpleAvatar`.

- AvataraAvatar
  The Avatara avatar provides a variety of additional functionality. The configuration is parsed by objects of the class `AvataraAvatar`. The Avatara package has to be previously installed in order to use the Avatar avatars.

Simple Avatar

A simple avatar is a very basic geometry with a texture representing the user in the virtual world. The simple avatars are not articulated and do not provide any features besides their static graphical representation.

- <name>
  The name of the avatar is provided in the `value` attribute of the <name>-node.
- <representation>
  This node encapsulates two sub-nodes describing the loaded model and the transformation of the model. It has been previously described in the introductory chapter.
  - <file>
    The <file>-node contains two attributes `type` and `name`. `Type` describes the file type of the model to be loaded. Currently 3DS and VRML are supported as file formats for simple avatars. In general this depends on the scene graph which is used for rendering, since its file loaders are incorporated to load the models. The `name` attribute describes the file name of the model which is to be loaded. Typically it is concatenated to the `Avatars` path setting in the general configuration.
  - <transformation>
    This node consists of four sub-nodes for <translation>, <rotation>,<scale>, and <scaleOrientation>as previously described in the introductory chapter.

```
<?xml version="1.0"?>
<!DOCTYPE simpleAvatar SYSTEM "http://dtd.invrs.org/simpleAvatar_v1.0a4.dtd">
<SimpleAvatar version="1.0a4">
  <name value="TieFigher" />
  <representation>
    <file type="VRML97" name="tie.wrl" />
    <transformation>
      <translation x="0" y="0" z="0" />
      <rotation x="0" y="1" z="0" angleDeg="0" />
      <scale x="1" y="1" z="1" />
    </transformation>
  </representation>
</SimpleAvatar>
```

Listing 2.5: exampleAvatar.xml

Avatara Avatar

The Avatara avatars are fully articulated avatars which provide features like support of animation, use of mesh skinning, and the control of specific bones like the spine, the head and the arm. A detailed introduction on the Avatara avatars is given in the *Avatara Manual*. The configuration of these avatars is performed in the class `AvataraAvatar`.

- <name>
  The name of the Avatara avatar.

- <representation>
  This node encapsulates two sub-nodes describing the loaded model and the transformation of the model.
  - <file>
    The <file>-node contains two attributes `type` and `name`. `Type` describes the file type of the model to be loaded. The type has to contain the string `"AvataraMDL"` which is a proprietary Avatara file format that can be exported from Blender, 3D Studio Max or Maya. The `name` describes the file name of the model. The extension of the files to be loaded is always `".mdl"`. Typically it is concatenated to the `Avatars` path setting in general configuration.
  - <transformation>
    This node consists of four sub-nodes for <translation>, <rotation>,<scale>, and <scaleOrientation>as previously described in the introductory chapter.
- <texture>
  This node defines an additional texture, which is to be mapped on the geometry. The file type of the texture has to be either set to JPEG, PNG or TGA.
- <animations>
  The <animations>-node contains the attributes `smooth` describing whether the transition between different animation phases should be smoothed or not, `speed` defining the animation speed, and `default` setting an initial animation sequence out of the following sub-nodes.
  - <animation>
    An <animation>-node describes an animation sequence with a given name identified by the attribute `name` and a path and file combination stored in `file` which identifies which animation file is to be used. As with the models of the Avatara package the animations use a proprietary file format with the extension `".ani"`. Animations can be exported as well from the modeling tools Blender, 3D Studio Max or Maya.

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

Listing 2.6: exampleAvataraAvatar.xml

2.3.2 Cursor Transformation Model

The available cursor models describe the behavior of the user's cursor. A variety of cursor models are already provided. Their configurations are parsed in the individual classes which are derived from `CursorTransformationModel`. Cursor models often work closely together with the configured interaction technique. The three standard cursor models are:

- VirtualHandCursorModel
- HomerCursorModel
- GoGoCursorModel

VirtualHandCursorModel

The most common cursor model used in VR applications is the virtual hand cursor model. The configuration is parsed in the objects of the class `VirtualHandCursorModel`. When using a virtual hand model a direct mapping between the gathered sensor data and the cursor is performed. The model should be used in conjunction with the virtual hand manipulation and confirmation model of the interaction module. It is set by a single node:

- <model>
  The `name` attribute of the <model>-node has to be set to `"VirtualHandCursorModel"`. The model does not take any additional arguments in the current implementation.

```
<?xml version="1.0"?>
<!DOCTYPE cursorTransformationModel SYSTEM "http://dtd.invrs.org/
    cursorTransformationModel_v1.0a4.dtd">
<cursorTransformationModel version="1.0a4">
  <model name="VirtualHandCursorModel"/>
</cursorTransformationModel>
```

Listing 2.7: exampleVirtualHandCursorModel.xml

HomerCursorModel

The **H**and-centered **O**bject **M**anipulation **E**xtending **R**ay-casting (HOMER) cursor model is designed to work with a setup of interaction transition functions used for an implementation of a HOMER interaction technique as described by Bowman and Hodges [BH97]. More detail on the functionality of the interaction technique can be found in their publication. The model is typically used with the HOMER selection action and the HOMER manipulation action model of the interaction module. The configuration is parsed in the objects of the class `HomerCursorModel`.

- <model>
  The `name` attribute of the <model>-node has to be set to `"HomerCursorModel"`. It does take three additional arguments.

This cursor model make use of the `ArgumentVector` as described in the introductory chapter. The possible arguments are:

- `animationSpeed`
  This attribute describes the movement speed of the cursor, when moving towards an object or back to the user representation.
- `forwardThreshold`
  This attribute is needed for collision detection during forward movement of the cursor. Collision between the cursor position and the target object are calculated. A common value would be 0.1.
- `backwardThreshold`
  The attribute is needed for collision detection during backward movement of the cursor. Typically the value is set to 0.1.

```
<?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="40"/>
      <arg key="forwardThreshold" type="float" value="0.1"/>
      <arg key="backwardThreshold" type="float" value="0.1"/>
    </arguments>
  </model>
</cursorTransformationModel>
```

Listing 2.8: exampleHomerCursorModel.xml

GoGoCursorModel

This cursor model is used in conjunction with the GoGo interaction technique as described by Poupyrev et al. [PBWI96]. It should be used in conjunction with the interaction models used for virtual hand interaction. The scaling is applied inside the cursor model. The configuration is passed down to the objects of the class `GoGoCursorModel`.

- <model>
  The `name` attribute of the <model>-node has to be set to `"GoGoCursorModel"`. The model takes two additional arguments.

This cursor model make use of the `ArgumentVector` as well. The possible arguments are:

- `distanceThreshold`
  This attribute sets a threshold, describing when to change from linear movement into an exponentially scaled movement.
- `k`
  This attribute is a constant value used for scaling the exponential distance calculation of the GoGo interaction technique.

```
<?xml version="1.0"?>
<!DOCTYPE cursorTransformationModel SYSTEM "http://dtd.invrs.org/
    cursorTransformationModel_v1.0a4.dtd">
<cursorTransformationModel version="1.0a4">
  <model name="GoGoCursorModel">
    <arguments>
      <arg key="distanceThreshold" type="int" value="20"/>
      <arg key="k" type="float" value="0.3"/>
    </arguments>
  </model>
</cursorTransformationModel>
```

Listing 2.9: exampleGoGoCursorModel.xml

2.3.3 User Transformation Model

These models describe the transformation of a user. In general the models are supposed to be implemented in classes derived from `UserTransformationModel`. Currently only a single model is implemented:

- HeadPositionUserTransformationModel

HeadPositionUserTransformationModel

This model maps the user position relative to the transformation which is gathered by the head sensor of the tracking system. The configuration of the model is parsed and evaluated by objects of the class `HeadPositionUserTransformationModel`.

```
<?xml version="1.0"?>
<!DOCTYPE userTransformationModel SYSTEM "http://dtd.invrs.org/
    userTransformationModel_v1.0a4.dtd">
<userTransformationModel version="1.0a4">
  <model name="HeadPositionUserTransformationModel">
</userTransformationModel>
```

Listing 2.10: exampelHeadPositionUserTransformationModel.xml

2.3.4 Cursor Representation

The cursor representation defines how the user cursor is represented in the VE. It is parsed and evaluated in objects derived from the class `CursorRepresentationInterface`. Depending on the configuration file used for the cursor representation a chosen type is selected. The available standard types for the cursor representation are:

- SimpleCursorRepresentation
- InteractionCursorRepresentation

SimpleCursorRepresentation

This type of cursor representation makes use of a simple model to be displayed at the cursors position. It is implemented in the class `SimpleCursorRepresentation`. A basic <representation>-node, similar to the one used for the representation of the simple avatars, is used for configuring the model.

```
<?xml version="1.0"?>
<!DOCTYPE simpleCursorRepresentation SYSTEM "http://dtd.invrs.org/
    simpleCursorRepresentation_v1.0a4.dtd">
<simpleCursorRepresentation version="1.0a4">
  <representation>
    <file type="VRML97" name="hand.wrl" />
    <transformation>
      <translation x="0" y="0" z="0" />
      <rotation x="0" y="1" z="1" angleDeg="0" />
      <scale x="0.5" y="0.5" z="0.5" />
    </transformation>
  </representation>
</simpleCursorRepresentation>
```

Listing 2.11: exampleSimpleCursorRepresentation.xml

InteractionCursorRepresentation

The interaction cursor representation is slightly advanced but still very basic. The implementation can be found in the class `InteractionCursorRepresentation`. It loads three models displayed differently based on the state of the interaction. The switching of the different models is issued by the interaction module.

- <idleModel>
  This model is displayed when the interaction state machine resides in the idle state. Indicating that no object is currently selected or manipulated.

- <selectionModel>
  The model is displayed when the interaction is in the selection state. An object has been selected for manipulation.

- <manipulationModel>
  The last model is displayed during the actual object manipulation.

```
<?xml version="1.0"?>
<!DOCTYPE interactionCursorRepresentation SYSTEM "http://dtd.invrs.org/
    interactionCursorRepresentation_v1.0a4.dtd">
<interactionCursorRepresentation version="1.0a4">
  <idleModel>
    <representation>
      <file type="VRML97" name="hand.wrl" />
      <transformation>
        <translation x="0" y="0" z="0" />
        <rotation x="0" y="1" z="1" angleDeg="0" />
        <scale x="0.5" y="0.5" z="0.5" />
      </transformation>
    </representation>
  </idleModel>
  <manipulationModel>
    <representation>
      <file type="VRML97" name="hand_closed.wrl" />
      <transformation>
        <translation x="0" y="0" z="0" />
        <rotation x="0" y="1" z="1" angleDeg="0" />
        <scale x="0.5" y="0.5" z="0.5" />
      </transformation>
    </representation>
  </manipulationModel>
</interactionCursorRepresentation>
```

Listing 2.12: exampleInteractionCursorRepresentation.xml

2.4 World Database

The world database contains information about the virtual world of an *inVRs* application. Three different nodes are used to configure the world database. The configuration of the database is triggered in the `WorldDatabase` object.
Similar to the user database this configuration file simply contains references to the different configurations of the individual components of the world database.

- <entityTypes>
  These objects act as templates for objects that are actually used for interaction in a VE. In this node a reference to a file containing the entity type database is given.

- <tiles>
  Tiles are mainly used for implementing the floor planes of a VE. They are designed to structure a VE in an easier fashion.

- <environmentLayout>
  The environment layout describes the alignment of the different environments, which again contain descriptions where the tiles and entities are located.

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

Listing 2.13: exampleWorldDatabase.xml

The following subsections introduce the individual components of the world database.

2.4.1 Entity Types

An entity is used typically as an interactive object in the environment. The configuration for the entities describes the different types of entities which can appear in an *inVRs* environment. The configuration of these entity types is parsed and evaluated in the corresponding class `EntityType`.

- <entityType>
  Entity types can be identified by a unique id stored in the attribute `typeId` and a name stored in the attribute `name`. Additionally it can be defined whether the entity can be used for interaction or whether it is static in the VE and can not be manipulated. This is determined by the boolean attribute `fixed`.
  - <representation>
    As for example with the avatars the representation contains information about the model with an initial transformation. With the entity types the `copy` attribute becomes of high importance. When the entities are finally instantiated in the environment configuration the geometry and texture data will be fully replicated[1] in case the `copy` attribute of the according entity type is set to true. Otherwise a reference to the model behind the entity is provided.

```
<?xml version="1.0"?>
<!DOCTYPE entityTypes SYSTEM "http://dtd.invrs.org/entityTypes_v1.0a4.dtd">
<entityTypes version="1.0a4">
  <entityType typeId="1" name="cube_1x1" fixed="1" >
    <representation copy="false">
      <file type="VRML" name="cube_1x1.WRL" />
    </representation>
  </entityType>
  <entityType typeId="10" name="CoordinateSystem" fixed="1" >
    <representation copy="false">
      <file type="VRML" name="coordinateSystem.wrl" />
    </representation>
  </entityType>
  <entityType typeId="20" name="inVRsLogo" fixed="0" >
    <representation copy="false">
      <file type="VRML" name="inVRsLogo.wrl" />
      <transformation>
        <translation x="0.0" y="0.7" z="0.0" />
        <scale x="0.3" y="0.3" z="0.3"/>
      </transformation>
    </representation>
  </entityType>
</entityTypes>
```

Listing 2.14: exampleEntity.xml

[1] implemented as deep clone in OpenSG

2.4.2 Tile

A tile can be used for defining a floor plane in an *inVRs* VE. Tiles have a rectangular shape and are a rather comfortable mechanism to layout a VE. The configuration of a tile is parsed and evaluated in the according `Tile` class.

- <tile>
  Tiles can be identified by a unique id, stored in the attribute `typeId` or a given name stored in the attribute `name`.
  - <tileProperties>
    The nodes <size>and <adjustment>describe the transformation of a given tile type used for layouting inside an environment. In the <size>-node the planar dimension of the representation which is only used for arranging a world is defined by the attributes `xSize` and `zSize`. These values do not affect the object representation of the tiles they are solely used for the arrangement and the layouting of the environment.
    Attributes of the <adjustment>-node do change the transformation of the tile. The `height` attribute allows for an additional translation along the y-axis. And the attribute `yRotation` stores a floating point value describing a positive rotation around the y-axis. The units for this rotation are degrees.
  - <representation>
    Similar to the entity types the representation contains information about the used model. The `copy` attribute is used in the same way considering the cloning of the displayed tiles.

```
<?xml version="1.0"?>
<!DOCTYPE tiles SYSTEM "http://dtd.invrs.org/tiles_v1.0a4.dtd">
<tiles version="1.0a4">
  <tile id="1" name="plane_10x10">
    <tileProperties>
      <size xSize="10" zSize="10" />
      <adjustment height="0" yRotation="0" />
    </tileProperties>
    <representation copy="false">
      <file type="VRML97" name="plane_10x10.WRL" />
    </representation>
  </tile>
</tiles>
```

Listing 2.15: exampleTile.xml

2.4.3 Environment and Environment Layout

The environments and their layout are used to describe the appearance and the arrangement of the virtual world. Previously defined entity types and tiles are placed and layouted in a VE inside these configuration files. The environment can be seen as a coordinate system or a region in a VE while the environment layout is used to arrange these environments.

Environment Layout

An <environmentLayout>-node configures the position of the different used environments on a plane. It can contain a single <tileGrid>-sub-node describing a grid resolution and additionally a set of environments which are to be aligned on this grid.

- <tileGrid>
  This node defines a grid inside the environment. It is used for the proper alignment of

tiles. The resolution of the grid is defined by the two attributes `xSpacing` and `zSpacing`. More detail on the definition of the grid layout is given in the description of the actual environments. No negative values are accepted for the spacing values.

- <environment>
  Many <environment>-nodes can exist. Each environment is identified by a unique id that is stored in the `id` attribute. It has an individual configuration file stored in the `name` attribute and a two-dimensional location defined in `xLoc` and `zLoc`. The location of the environment is finally dependent on the `xSpacing` and `zSpacing` attributes of the <tileGrid>-node as well as the `xLoc` and `zLoc` attributes of this node.
  To determine in world coordinates where the upper left corner of an environment is located, one can simply multiply the tile grids spacing and the loc attributes of the environment layout. The loc attributes can contain negative values.

```
<?xml version="1.0"?>
<!DOCTYPE environmentLayout SYSTEM "http://dtd.invrs.org/environmentLayout_v1.0a4.
    dtd">
<environmentLayout version="1.0a4">
  <tileGrid xSpacing="10" zSpacing="10"/>
  <environment id="1" name="environment.xml" xLoc="0" zLoc="0"/>
</environmentLayout>
```

Listing 2.16: exampleEnvironmentLayout.xml

Environment

An environment can be considered as a certain region of an *inVRs* VE or as an individual sub-coordinate system. It does not have a visual representation. In the configuration the actual placement of entities as well as tiles takes place. The coordinate system of an environment has its origin in the top left corner environment. The configuration of an environment is parsed and evaluated in the according `Environment` class.

- <tileMap>
  The <tileMap>-node is used for the specification of a map of tiles. It takes two arguments defining the dimension of the map `xDimension` and `zDimension` which describe the size of a 2D grid. These dimension attributes multiplied with the spacing attributes of the environment layout configuration define the size of an environment in world coordinates.
  The size of a tile defined in the `xSize` and `zSize` attributes of the <tile>-node is taken into account for layouting. Inside the node a list of tiles is inserted. This list contains the ids of the tiles.

- <entryPoint>
  This node describes an entry point used at startup in a VE. Camera and user transformation is typically set to this entrypoint described by the six attributes `xPos`, `yPos`, `zPos`, `xDir`, `yDir` and `zDir`. The transformation of an entry point is to be seen in local environment coordinates.
  In general it is possible to use a whole list of entry points, which can be processed by the application.

- <entity>
  Basically an instance of an entity of a certain entity type is provided by nodes of this type. The attribute `id` refers to a unique id while the attribute `typeId` refers to its entity type which has to be previously defined in an <entityType>-node in the entity types configuration.
  Additionally the sub-node <transformation>can be used to alter the transformation of this specific entity. First the entity type transformation is applied and afterwards the actual

entity transformation is applied. This transformation is local in environment coordinates of the environment where the entity is located.

```
<?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="5" yPos="4" zPos="-5" xDir="0" yDir="0" zDir="1"/>

  <entity id="1" typeId="1">
    <transformation>
      <translation x="3.5" y="0" z="3.5"/>
      <rotation x="0.00" y="1.00" z="0.00" angleDeg="0"/>
      <scale x="1.00" y="1.00" z="1.00"/>
    </transformation>
  </entity>

  <!-- TrackedHeadTransformation -->
  <entity id="23" typeId="10">
    <transformation>
      <translation x="5" y="1" z="5"/>
      <rotation x="0.00" y="1.00" z="0.00" angle="0"/>
      <scale x="0.50" y="0.50" z="0.50"/>
    </transformation>
  </entity>

  <!--  inVRs logo -->
  <entity id="30" typeId="20">
    <transformation>
      <translation x="5" y="0" z="9.5"/>
      <rotation x="0.00" y="1.00" z="0.00" angle="180"/>
      <scale x="1" y="1" z="1"/>
    </transformation>
  </entity>
</environment>
```

Listing 2.17: exampleEnvironment.xml

2.5 Event Manager

A detailed introduction on transformation and event handling in the *inVRs* framework is given in [ALBV07]. The event manager so far provides a very basic setup which is stored in source code. It does not have any additional configuration capabilities yet, which is likely to change in newer versions of the framework. Thus this section has been integrated to support extension and for completeness sake.
The implementation of the event manager can be found in the class `EventManager`.

2.6 Transformation Manager

The configuration of the transformation manager might seem fairly uncomfortable at first sight but in the end it is pretty powerful. The configuration files are parsed and evaluated by the class `TransformationManager`. The main components of a transformation manager setup are pipes and mergers. Inside the pipes modifiers have to be set up which alter the transformations flowing through the pipes. Mergers can be used to combine the results of two or more pipes.
More detail on transformation management and pipe setup is provided in [ALBV07].

- <logFile>
  This node takes a single attribute `name` the value of the attribute describes the name of the log file where the information generated by a potential logging modifier is written to. Such log files could be used for replay purposes or debugging.

- <pipeList>
  This node contains a list of sub-nodes of the type pipe. Each object or entity in the VE which is to be transformed and makes use of the transformation manager should have a pipe opened.

  - <pipe>
    A pipe is used to route transformations from one *inVRs* component to another *inVRs* component. A pipe has a complex set of attributes. They are implemented in the class `TransformationPipe`. The source and the target component identifying from where to where the transformations are to be routed are described by the two attributes `srcComponentName` and `srcComponentName`.
    With the attribute `pipeType` it is possible to define a specific type of a pipe. The value Any is used as a wildcard. The attributes `objectClass`, `objectType`, and `objectId` are highly dependent on the component, where the pipe was opened. Wildcards can be used there as well.

    * <modifier>
      The modifiers inside a pipe are executed sequentially on the transformations flowing through the pipe. The are processed in the same order as they are set up in the configuration. An overview and a detailed description on the so far available modifiers is given in Section 2.6.1.

- <mergerList>
  The concept of mergers was introduced to ease the implementation of concurrent object manipulation. This node contains several sub-nodes each describing a merger. Once two different pipes are opened on the same object in the transformation manager this conflict is detected and can be resolved by using a merger.

  - <merger>
    One node in a list of mergers is able to combine the results of two different pipes and write it to a new output pipe. The are implemented as sub-classes of the class `TransformationMerger`.

    * <inputPipe>
      The attributes of an <inputPipe>-node are identical to a normal pipe. In case the merger has detected two sources accessing a pipe with the same attributes as the input pipe it becomes active.

    * <outputPipe>
      The attributes of an <outputPipe>-node are identical to a normal pipe. If a merging process has taken place the results are written on this output pipe.

```
<?xml version="1.0"?>
<transformationManager>
  <!-- srcModule: ID as defined in ModuleIds.h (USER_DEFINED_ID == 0) -->
  <pipe srcModuleName="NavigationModule" dstModuleName="TransformationManager"
    pipeType="Any" objectClass="Any" objectType="Any" objectId="Any"
    fromNetwork="0">
    <modifier type="ApplyNavigationModifier" />
    <modifier type="HeightMapModifier" />
    <modifier type="CheckCollisionModifier">
      <arguments>
        <arg key="radius" type="float" value="1" />
        <arg key="fileName" type="string" value="MedievalTownCollisionMap.wrl"/>
```

```
      </arguments>
    </modifier>
    <modifier type="TransformationDistributionModifier" />
    <modifier type="UserTransformationWriter" />
    <modifier type="CameraTransformationWriter" >
      <arguments>
        <arg key="cameraHeight" type="float" value="1.8"/>
        <arg key="useGlobalYAxis" type="bool" value="true"/>
      </arguments>
    </modifier>
    <modifier type="AvatarTransformationWriter" >
      <arguments>
        <arg key="clipRotationToYAxis" type="bool" value="true" />
      </arguments>
    </modifier>
    <modifier type="ApplyCursorTransformationModifier" />
    <modifier type="CursorTransformationWriter" />
  </pipe>
  <pipe srcModuleName="InteractionModule" dstModuleName="WorldDatabase"
    pipeType="Any" objectClass="Any" objectType="Any" objectId="Any"
    fromNetwork="0">
    <modifier type="ManipulationOffsetModifier"/>
    <modifier type="TransformationDistributionModifier"/>
    <modifier type="EntityTransformationWriter" />
  </pipe>
  <pipe srcModuleName="InteractionModule" dstModuleName="WorldDatabase"
    pipeType="Any" objectClass="Any" objectType="Any" objectId="Any"
    fromNetwork="1">
    <modifier type="EntityTransformationWriter" />
  </pipe>
  <pipe srcModuleName="NavigationModule" dstModuleName="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>
</transformationManager>
```

Listing 2.18: exampleTransformationManager.xml

2.6.1 Modifiers

In general there is a fair bit of modifiers available already which will be explained in the following sub-sections. In a more fine grained way these different modifiers can be categorized in three main types:

- Modifier
  A general modifier which typically alters the transformation passing through the pipe.
- Writer
  These special modifiers do not alter the content of the transformation stored in the pipe. They simply write data to an external component, which does not have to be the target component.
- Interrupter
  Interrupters are used stop the further processing of a pipe.

Many of them are implemented using argument vectors. The modifiers are all derived from the class `TransformationModifier`.

They are placed as a pipe stage inside a pipe with the node name <modifier>and carry the attribute `type`. This attribute has to be set to a valid modifier name as listed in the following:

ApplyNavigationModifier

This modifier is implemented in the class `ApplyNavigationModifier`. It is used in a navigation pipe and applies the transformation currently in the pipe, which should have been received from the navigation module, on the navigated transformation currently stored in the user object.
The resulting transformation is then written in the pipe for further processing. The modifier takes the following arguments:

- `centerAtWorldUser`
  This optional boolean variable defines the point of rotation. It is set to true (default) then the rotation change is calculated around the world user transformation, otherwise the rotation change is applied directly to the navigated transformation.

HeightMapModifier

The height map modifier is parsed in the class of the same name `HeightMapModifier`. An additional height map is used to alter the current transformation stored in the pipe. To get details on the use of height maps it is helpful to work through the *Medieval Town Tutorial*. Details on the generation and processing of the *inVRs* heightmaps are given in [BLAV06]. The modifier takes the following arguments:

- `fileName`
  This variable defines the heightmap file which should be loaded.
- `scale`
  This optional float value defines a scale value which scales the height values of the heightmap.
- `offset`
  This optional float value allows to define a height offset which is added to the height values of the heightmap.

CheckCollisionModifier

The check collision modifier is parsed in the class of the same name `CheckCollisionModifier`. An additional collision map is used to correct the given transformation. It makes use of an additional argument vector which can take the following parameters:

- `radius`
  This attribute describes the radius of a circle around the transformation in the pipe. The collision map is checked against this circle. In case a collision has taken place the transformation is corrected, otherwise it is just passed through.
- `fileName`
  This attribute stores the collision map against which the circle is to be checked. It is a simple VRML file containing collision lines for the VE.

Information on the creation and usage of collision maps can be found in the *Medieval Town Tutorial* as well as in [BLAV06].

TransformationDistributionModifier

This modifier is parsed in the class of the same name `TransformationDistributionModifier`. The given transformation in the pipe is written to the network module in order to be distributed to the interconnected participants. At the remote site it will be inserted again in a pipe of the remote transformation manager.

- `protocol`
  The transmission protocol has to be set. The supported values are UDP (default) and TCP.

UserTransformationWriter

This modifier is parsed in the class of the same name `UserTransformationWriter`. It is used to set the base position of the user representation in the VE. The modifier takes the following arguments:

- `useLocalUser`
  This optional boolean argument allows to define that the transformation should be written to the local user instead of the owner of the pipe (which is the default behavior). This can be used to change the user's transformation from pipes owned by another user.

CameraTransformationWriter

This modifier is as well parsed in the class of the same name `CameraTransformationWriter`. The writer is used for setting up the camera transformation. It makes use of an additional argument vector which can take the following parameters:

- `cameraHeight`
  This optional float attribute adds an additional offset to the height of the camera. It can be useful if for example terrain following mechanisms are implemented which make use of height maps in order to raise the camera above the terrain level.

- `useGlobalYAxis`
  In case this optional boolean attribute is set to `true` the offset defined in `cameraHeight` is applied on the y-axis of the global coordinate system instead of the local coordinate system of the user transformation.

- `useLocalUser`
  This optional boolean argument allows to define that the transformation should be written to the camera of the local user instead of the owner of the pipe (which is the default behavior).

AvatarTransformationWriter

This modifier is parsed in the class of the same name `AvatarTransformationWriter`. This writer sets the transformation used for the user representation. It makes use of an additional argument vector which can take the following parameter:

- `useLocalUser`
  This optional boolean argument allows to define that the transformation should be written to the avatar of the local user instead of the owner of the pipe (which is the default behavior).

- `clipRotationToYAxis`
  By setting this optional argument to `true` the user representation is always displayed in an upward position. It is still possible to change the orientation of the camera, but the avatar will always stay upwards.

ApplyCursorTransformationModifier

This modifier is parsed in the class of the same name `ApplyCursorTransformationModifier`. It additionally applies the cursor transformation in combination with the selected cursor transformation model. The modifier takes the following arguments:

- `useLocalUser`
  This optional boolean argument allows to define that the cursor transformation of the local user should be calculated instead of the cursor transformation of the pipe owner (which is the default behavior).

CursorTransformationWriter

This modifier is parsed in the class of the same name `CursorTransformationWriter`. It is used to write the transformation of the users cursor. This modifier is dependent on the chosen cursor model. The modifier takes the following arguments:

- `useLocalUser`
  This optional boolean argument allows to define that the transformation should be written to the cursor transformation of the local user instead of writing the cursor transformation of the pipe owner (which is the default behavior).

ManipulationOffsetModifier

This modifier is parsed in the class of the same name `ManipulationOffsetModifier`. During interaction it is often common that an offset from the point where the object was picked and the origin of the object has to be applied. This modifier takes care of this offset. The modifier supports no arguments.

EntityTransformationWriter

This modifier is parsed in the class of the same name `EntityTransformationWriter`. It is mainly used in pipes for interaction where the entity is transformed as a result of manipulation. The writer updates the transformation of the entity. The target entity is determined by the ID of the transformation pipe. The writer supports no arguments.

TrackingDataWriter

If a tracking system is used this writer can be used to write the transformation from the pipe to the transformation of the corresponding tracking data transformation in the user object. This modifier is parsed in the class of the same name `TrackingDataWriter`. The writer supports no arguments.

TrackingOffsetModifier

This modifier is parsed in the class of the same name `TrackingOffsetModifier`. It allows to add an offset from a sensor from a tracking system to the incoming transformation. The modifier takes the following arguments:

- `useHeadSensor`
  This optional boolean argument defines if the Head sensor or the Hand sensor should be used.

- `removeYAxis`
  This optional boolean argument allows to define that the sensor position in the Y-axis (upwards) should be removed before applying the offset.

- `removeOrientation`
  This optional boolean argument allows to define that the orientation should not be applied as offset.

- `useLocalUser`
  This optional boolean argument defines whether the tracking data information should be read from the owner of the pipe (default behavior) or the local user.

TransformationLoggerModifier

To allow for detailed transformation logging, which might be used for later replay purposes a logging modifier is provided. The logfile which is used is defined in the `logfile` element. This modifier is parsed and evaluated in the class of the same name `TransformationLoggerModifier`. The modifier takes the following arguments:

- `policy`
  This optional argument allows to define a logging policy. Possible values are `iterationbased` (default) and `timebased`.

- `policyParam`
  Depending on the selected logging `policy` the argument defines either the number of iterations between two log file entries (`iterationbased`) or the number of seconds between to log file entries (`timebased`)

TargetPipeTransformationWriter

This modifier allows to write the incoming transformation into another transformation pipe. This modifier is parsed in the class of the same name `TargetPipeTransformationWriter`. The modifier takes the following arguments:

- `srcId`
  Source component ID of the target pipe.

- `dstId`
  Destination component ID of the target pipe.

- `pipeType`
  Type of the target pipe.

- `objectClass`
  Object class of the target pipe.

- `objectType`
  Object type of the target pipe.

- `objectId`
  Object Id of the target pipe.

- `fromNetwork`
  Defines if the target pipe is from network or not.

MultiPipeInterrupter

This modifier allows to interrupt all pipes in which it is registered at once. This modifier is parsed in the class of the same name `MultiPipeInterrupter`. The modifier takes no arguments.

AssociatedEntityInterrupter

The AssociatedEntityInterrupter stops the execution of the transformation pipe if the `objectType` and the `objectId` fields of the pipe ID match to one of the associated entities of the local user. This is the case when a user is manipulating an entity. This modifier is parsed in the class of the same name `AssociatedEntityInterrupter`. The modifier takes no arguments.

2.6.2 Mergers

Mergers are used if two pipes point to the same target. They take the attribute `type` which has to be one of the following list. Mergers are implemented in classes derived from the class `TransformationMerger`.

SharedManipulationMerger

The SharedManipulationMerger can be used in order to allow multiple users to interact with a single Entity at the same time. It therefore merges the manipulation pipes of those users to a single pipe. In this pipe the resulting transformation can then be written to the entity using the `EntityTransformationWriter`. The merger is implemented in the class of same name `SharedManipulationMerger`. The modifier takes no arguments.

2.7 Summary

This chapter has first introduced the use of the general configuration, providing an initial setup, links to the setup of the *inVRs* components were given and the path settings of the framework were defined.
In the following section the setup of the system core and its individual components was explained in detail. The user database used for managing user data, user representation and the transformations for the user and cursor were described.
The world database configuration which used for setting up the virtual world with the entities, tiles and environments has been explained in detail.
Two managers manage the communication of the *inVRs* core with its components. The basic implementation of the event manager does not provide any configuration mechanisms but it was introduced for the sake of completeness and extension.
The next manager the transformation manager is a highly configurable sub-component of the core. Transformation data is passed through pipes which are configured by setting up pipe stages or modifiers. This chapter has listed all the standard modifiers that are used for the configuration of the transformation manager. The mergers used for combination of the pipes were listed and described.
The following chapters introduce the interface configurations and module configurations.

Chapter 3

Input Interface

The input interface of *inVRs* is designed to handle arbitrary input devices or input types. Examples would be tracking systems, wands, mice, keyboards, joysticks or input generated by arduino boards [1]. The implementation so far takes care of an abstract controller that represents a virtual input device which is then accessed in a standardized way by the *inVRs* components.
This controller is handled by a controller manager. The interface is parsed and evaluated by the `InputInterface` class. The actual manager is handled by the `ControllerManager` class.
The configuration of the <module>-node providing a link to the controller takes up to three attributes:

- `name`
  This attribute describes the name of the interface to be included for input. To include the controller used for processing of standard input devices use the string `"ControllerManager"` as input value.
- `configFile`
  This setup file describes name of the configuration file for the controller. It is concatenated to the <path>-node with the attribute `name` set to `"ControllerConfiguration"` as defined in the general configuration.
- `libraryName`
  In case the name of the library differs to the input file the name of the library can be provided separately.

In general it is of course possible to interconnect other input mechanisms to the input interface, like for example speech processing and speech commands or gesture recognition, which is not support by the framework yet.

```
<?xml version="1.0"?>
<!DOCTYPE inputInterface SYSTEM "http://dtd.invrs.org/inputInterface_v1.0a4.dtd">
<inputInterface version="1.0a4">
  <module name="ControllerManager" configFile="MouseKeybSensorController.xml"/>
</inputInterface>
```

Listing 3.1: exampleInputInterface.xml

3.1 Controller Manager

The controller manager is responsible for handling abstract input devices which consist of components of a set of physical devices. Inside a <controller>-node one of such devices is defined.

[1] http://www.arduino.cc/

In general it is possible to use only a single controller. This node is used to define the amount of abstract components of which the abstract controller consists of. It is followed by an arbitrary number of <device>-nodes which perform the mapping of the components of single or several physical devices on the components of the abstract controller.

- <controller>
  This node with its sub-nodes describes the complete data mapping for a single controller it can contain the following three attributes. The attribute `axes` describes how many axes the controller finally should contain. In `buttons` it is defined how many button the controller should contain, and finally `sensors` defines the amount of sensors inside the newly defined controller.
  - <device>
    Different types of supported physical devices or device libraries exist which will be explained in the later sub-sections. This node takes one argument `type` describing the name of device based on which the appropriate driver is selected. Devices can have an argument vector describing their specific setup. Depending on the driver or library each device can have an arbitrary amount of sub-nodes of the following three types.
    * <button>
      This node takes two attributes. One of the type `deviceIndex` which describes the driver based id of the button and one of the type `controllerIndex` describing the abstract id of the button. Additionally this node can contain a sub-node <buttonCorrection>which takes the attribute `invert` describing whether the button value should be inverted. Typically a button delivers the value 1 if pressed and 0 otherwise.
    * <axis>
      Similar to the <button>-node this node has to take the attributes `deviceIndex` and `controllerIndex` for mapping purposes. Additionally two more attributes, describing thresholds for the axis, `minValue` and `maxValue` can be set in a sub-node called <axisValues>. They can contain positive as well as negative integer or floating point values. The <axis>-node can contain an additional sub-node called <axisCorrection>. This node takes the two attributes `scale` and `offset`.
    * <sensor>
      Like the <button>-node and the <axis>-node this node has to take the attributes `deviceIndex` and `controllerIndex` for mapping purposes. Additionally it can contain several nodes for adjustment. The <coordinateSystemCorrection>-node which is similar to a common <transformation>-node but omitting the scale orientation attribute is used for modifying a sensor transformation in general. Another sub-node the <positionCorrection>-node takes the attributes `translation` and `scale` in order to perform additional fine tuning. The same approach is used by the additional <orientationCorrection>-node which uses `rotation` as an argument.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE controllerManager SYSTEM "http://dtd.invrs.org/controllerManager_v1.0a4.
    dtd" >
<controllerManager version="1.0a4">
  <controller buttons="2" axes="1" sensors="1">
    <device type="VrpnDevice">
      <arguments>
        <arg key="" value="" />
      </arguments>
      <button deviceIndex="0" controllerIndex="0">
        <buttonCorrection invert="true" />
      </button>
      <button deviceIndex="1" controllerIndex="1" />
      <axis deviceIndex="0" controllerIndex="0">
```

```
        <axisValues minValue="-513" maxValue="11378" />
      </axis>
      <axis deviceIndex="1" controllerIndex="1" />
      <sensor deviceIndex="0" controllerIndex="0">
        <coordinateSystemCorrection>
          <translation x="0" y="0" z="0" />
          <rotation x="" y="" z="" />
          <scale x="" y="" z="" />
        </coordinateSystemCorrection>
        <positionCorrection>
          <translation x="" y="" z="" />
          <scale x="" y="" z="" />
        </positionCorrection>
        <orientationCorrection>
          <rotation x="" y="" z="" />
        </orientationCorrection>
      </sensor>
      <sensor deviceIndex="1" controllerIndex="1" />
    </device>
  </controller>
</controllerManager>
```

Listing 3.2: exampleControllerManager.xml

3.1.1 Supported Devices

The *inVRs* framework supports so far four different types of devices. It makes use of a variety of external libraries in order to abstract input. Right now GLUT [2], trackD[3], VRPN[4] [THS+01] and a so called UDP device are supported. Developing your own device is straight forward and is described in detail in the *Going Immersive Tutorial*.
Most devices support buttons, axes and sensors. Some of them can be configured specifically by using the argument vectors. The following subsections introduce the available devices an describe their additional arguments if available.

GlutCharKeyboardDevice

The GlutCharKeyboardDevice takes input from the GLUT keyboard function. It is implemented in the class `GlutCharKeyboardDevice`. The keyboard supports the use of buttons. When using OpenSG as a scene graph GLUT is typically already installed.

GlutMouseDevice

The GlutMouseDevice takes input from the GLUT mouse function. This device makes additional use of an argument vector. It is implemented in the class `GlutMouseDevice`. These devices make use of axes, representing the 2D mouse coordinates and additional wheel input, as well as buttons.

- `axisReleaseSpeed`
  A value which has to be set for fine tuning the mapping from the mouse on an axis. If the value is chosen to low the movement might appear bumpy. In case it is to high the axis update might lag behind. A common value for this attribute is 20.

GlutSensorEmulatorDevice

This device is used to emulate 3D tracking sensors with ordinary GLUT input. The implementation can be found in the class `GlutSensorEmulatorDevice`. It is used to emulate a 6DOF or several

[2] http://www.opengl.org/resources/libraries/glut/
[3] http://www.mechdyne.com/integratedSolutions/software/products/trackd/trackd.htm
[4] http://www.cs.unc.edu/Research/vrpn/

6DOF sensors by using input from conventional desktop devices like mice and keyboards. This device makes additional use of an argument vector. The possible arguments of that device are:

- `numberOfSensors`
  Describes the amount of sensors to be emulated.
- `switchSensorButton`
  A button id, which toggles between the different sensors.
- `switchTransformationTargetButton`
  A button id, which is used to toggle between the transformation targets. Transformation targets are either translation or rotation of the current sensor.
- `switchAxesButton`
  A button id, which toggles between the access control of a sensor axis.

UdpDevice

Often it becomes necessary to abstract input even more, for example when your input device is not directly connected to your machine running the *inVRs* application. This device communicates with an external input server. It is implemented in the class `UdpDevice`.
Additionally the server has to be set up which communicates with the UDP device. These server are normally very specific to devices and have to be implemented by the user. Examples for the server are provided in the *inVRs* source distribution.

TrackdDevice

The support of trackD devices has to be added externally. The implementation can be found in the class `TrackdDevice`. TrackD as a library makes use of shared memory segments in which the data of the controller (buttons and joystick) as well as the data of the tracker (sensor data) are written.
This device makes additional use of an argument vector. The possible arguments of that device are:

- `controllerKey`
  The id of the shared memory segment to which the controller data is to be written. The data consists of axes and buttons.
- `trackerKey`
  The id of the shared memory segment to which the data of the tracking system is written to. The data consists of 6DOF sensors.

VrpnDevice

Alternatively to trackD the open source library VRPN can be used to preprocess input data. This library has to be additionally installed. It is implemented in the class `VrpnDevice`. VRPN supports buttons, axes and 6DOF sensors.

- `deviceID`
  The id of the VRPN tracking server e.g. `iotracker@140.78.198.157`
- `numSensors`
  An integer value describing the amount of used sensors.
- `numButtons`
  Describes the amount of used buttons.
- `numAxes`
  The number of axes.

3.2 Summary

This chapter has given an overview of the configuration of the input interface of the *inVRs* framework. So far input from arbitrary input devices is handled by the controller manager. This manager abstracts and exposed data gathered from the libraries GLUT, trackD, VRPN and data provided from the network. The configuration of the different libraries for the use with *inVRs* has been explained in detail.

Chapter 4

Output Interface

The output interface of *inVRs* is designed to support graphical output on a scene graph level and audio output with the help of external audio libraries. The configuration of the output interface contains references to the chosen module. It is parsed and evaluated by the `OutputInterface` class.
So far this class is very basic and simply provides a reference to the used scene graph interface. The `OpenSGSceneGraphInterface` is implemented following the same architecture as a module and thus acts as a plugin for the `OutputInterface`. The configuration takes up to three attributes:

- `name`
  This attribute describes the name of the interface to be included for output.
- `configFile`
  This setup file describes name of the configuration file for the output interface.
- `libraryName`
  In case the name of the library differs to the input file the name of the library can be provided separately.

```
<?xml version="1.0"?>
<!DOCTYPE outputInterface SYSTEM "http://dtd.invrs.org/outputInterface_v1.0a4.dtd">
<outputInterface version="1.0a4">
  <module name="OpenSGSceneGraphInterface" configFile=""/>
</outputInterface>
```

Listing 4.1: exampleOutputInterface.xml

4.1 OpenSG Scene Graph Interface

Currently no specific configuration file is available for the OpenSG[1] scene graph interface. It is mainly used for extension. By supporting OpenSG as a scene graph it is easy to incorporate all types of multi-display systems like CAVEs [CNSD+92] or curved installations like the i-Cone [SG02].
A reasonable additional configuration could be the initial setup of graph operators.

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

4.2 OpenSceneGraph Scene Graph Interface

To allow for scene graph independence of the framework the interface to OpenSceneGraph[2] is currently under development.

4.3 Audio Interface

The interface to the OpenAL[3] audio library is currently under development. The interface will provide a list of sound files which will be loaded initially and can be replayed from inside an *inVRs* application.

4.4 Summary

This chapter has briefly introduced the interface to possible scene graphs and audio libraries. It is likely that the chapter will be extended soon, since an implementation of the interface to OpenSceneGraph as well as OpenAL is currently under development.

[2] http://www.openscenegraph.org/
[3] http://www.openal.org/

Chapter 5

Navigation Module

The configuration of the navigation module always contains three different nodes which are used to set up the three types of navigation models. It is parsed by the `Navigation` class and successively by the individual models that are derived from the classes `OrientationModel`, `TranslationModel` and `SpeedModel`.

- <translationmodel>
  The translation model describes in which direction the movement of the camera or the object attached to the navigation module takes place. A variety of models exist which will be explained in Section 5.1.1. The parameter for this node is `type` describing the name of the chosen model. Further configuration which is model specific can be found in the argument vector.

- <orientationmodel>
  The orientation model describes what orientation is to be kept during the navigation. As with the translation models a variety of models exist which are explained in detail in Section 5.1.2. The parameter of the orientation model is `type` describing the name of the model. Additionally a value for the parameter `angle` has to be parsed defining the speed of the orientation change. All further configuration is dependent on the chosen model and is described in a model specific argument vector.

- <speedmodel>
  This model defines the speed of the navigation. Besides the name of the model to be used indicated by the parameter `type` the speed model takes an additional parameter. The second parameter `speed` scales the speed which is to be applied by the speed model. A list of possible speed models is provided in Section 5.1.3. As the other transformation models the speed models make use of the argument vector.

The following configuration shows an example of a complete navigation setup. When using the navigation module it is important to check the compatibility in between the individual models. One has to be careful as well when the models are exchanged or when the input controller is exchanged.

```
<?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="frontIdx" type="uint" value="3"/>
      <arg key="backIdx" type="uint" value="4"/>
      <arg key="leftIdx" type="uint" value="5"/>
      <arg key="rightIdx" type="uint" value="6"/>
    </arguments>
```

```
  </translationModel>
  <orientationModel type="OrientationDualAxisModel" angle="10">
    <arguments>
      <arg key="xAxisIdx" type="int" value="0"/>
      <arg key="yAxisIdx" type="int" value="1"/>
      <arg key="buttonIdx" type="int" value="1"/>
    </arguments>
  </orientationModel>
  <speedModel type="SpeedMultiButtonModel" speed="5">
    <arguments>
      <arg key="accelButtonIndices" type="string" value="3 4 5 6"/>
    </arguments>
  </speedModel>
</navigation>
```

Listing 5.1: exampleNavigationModule.xml

5.1 Navigation Models

A large set of independent models is already available. Details on the models used for navigation with the theoretical background can be found in [AHKV04].
All models can be set independently. The provided models do not influence other models in any way, meaning the description on how the orientation is to be set is for example independent from the movement direction, unless the same input channels are used.

5.1.1 Translation Models

All used translation models have to be derived from the class `TranslationModel`. A translation model describes the direction of the resulting matrix of the navigation module. Model specific parameters are provided by an argument vector.

TranslationViewDirectionModel

This model is implemented in the class `TranslationViewDirectionModel`. The movement direction is always identical to the view direction when this translation model is used.

TranslationViewDirectionButtonStrafeModel

This model makes use of four different buttons in order to indicate the movement direction in combination with the view direction. The implementation of the model can be found in the class `TranslationViewDirectionButtonStrafeModel`. It is possible to set the translation in the view direction, negative to the view direction and to the left or the right of the view direction, by pressing the according buttons.

- `frontIndex`
  The id of the button used to move in the view direction.
- `backIndex`
  The id of the button used to move in the negative view direction.
- `leftIndex`
  The id of the button used to strafe left from view direction.
- `rightIndex`
  The id of the button used to strafe right from view direction.

TranslationViewDirectionAxisStrafeModel

The class `TranslationViewDirectionAxisStrafeModel` is responsible for parsing and evaluating the given model. With this model it is possible to set the translation in the view direction, negative to the view direction and to the left or the right of the view direction, by pressing the according axis in a positive or negative direction.

- `leftRightIndex`
  The index of the axis used for left and right translation relative to the view direction.
- `frontBackIndex`
  The index of the axis used for front and back translation relative to the view direction.

TranslationSensorViewDirectionModel

The model is designed for the use with immersive displays incorporating position tracking. It makes uses of a 6DOF sensor. The implementation of the model can be found in the class `TranslationSensorViewDirectionModel`. The translation vector points initially in the view direction. It is adjusted by adding the orientation of the used sensor.

- `sensorIndex`
  The index of the sensor to be used.
- `ignoreYAxis`
  In case this boolean attribute is set to true, the y-axis of the sensor is ignored.

5.1.2 Orientation Models

The orientation models are used to define the orientation during navigation. They are all derived from the class `OrientationModel`. Model specific parameters are provided by an argument vector.

OrientationSingleAxisModel

The model can be used for changing the orientation around the Y-axis via a single axis value. A use case for this model could be a navigation mode realizing walking on a 2-dimensional flat terrain. This orientation model is implemented in `OrientationSingleAxisModel`.

- `axisIndex`
  The index of the axis to be used.
- `minThreshold`
  The minimum axis value which is taken as input (values below this threshold are ignored)

OrientationDualAxisModel

This model makes use of two axis values for input and a button value. It can be used to control the orientation around all three axes by a combination of these two controller axes in combination with a controller button. One controller axis (defined by argument yAxisIndex) always controls the rotation around the local X-axis (which corresponds to rotation upwards/downwards). The second controller axis (defined by argument xAxisIndex) can be used to change the orientation around the local Y-axis (left/right) or the local Z-axis (clockwise/counterclockwise). The destination rotation axis is defined by the controller button value (defined by argument buttonIndex). If the button is pressed then the Z-axis is used, otherwise the rotation around the Y-axis is executed. A common use case for this model is when a mouse device is used for input. This model is implemented in the class `OrientationDualAxisModel`.

- `xAxisIndex`
  The index of the axis to be used for rotation around the Y-axis (if button is not pressed) or Z-axis (if button is pressed).
- `yAxisIndex`
  The index of the axis to be used for rotation around the x-axis.
- `buttonIndex`
  The index of the button which has to be pressed for changing the target rotation axis.

OrientationButtonModel

The orientation in this model is designed to be used with six different buttons. Each of the buttons is used change the rotation around one of the three rotation axes either in positive or negative direction. It is implemented by the class `OrientationButtonModel`.

- `downIndex`
  The index of the button used for downward rotation.
- `upIndex`
  The index of the button used for upward rotation.
- `leftIndex`
  The index of the button used for left rotation.
- `rightIndex`
  The index of the button used for right rotation.
- `cwIndex`
  The index of the button used for clockwise rotation.
- `ccwIndex`
  The index of the button used for counter clockwise rotation.

OrientationSensorModel

This model is implemented in the class `OrientationSensorModel`. When this model is used the orientation is changed according to the orientation of a single sensor. It makes use of an argument vector and can the the following parameters:

- `sensorIndex`
  The index of the sensor to be used.
- `minThreshold`
  The minimum rotation angle for each axis to be taken into account
- `mirrorAdjXFactor`, `mirrorAdjYFactor`, `mirrorAdjZFactor`
  Optional arguments for mirroring the rotation axes of the sensor orientation
- `rotationAdjAxisX`, `rotationAdjAxisY`, `rotationAdjAxisZ`, `rotationAdjAngleDeg`
  Optional arguments for correcting the rotation of the controller sensor orientation input

5.1.3 Speed Models

All speed models are derived from the class `SpeedModel`. They are used to describe the movement speed during the navigation. As with the orientation and translation models all model specific parameters are provided by an argument vector.

SpeedFixedSpeedModel

This model is the simplest implementation of a speed model. It is implemented in the the class `SpeedFixedSpeedModel`. A constant speed value is taken from the initial configuration of the navigation stored inside the `speed` attribute of the <speedmodel>-node.

- `useTimestep`
  Optional argument which defines if the timestep should be multiplied (which is done be default) or not. This can be useful when the outcome of the navigation should be used as input for some timestep independent values, like the force which should be applied to an object.

SpeedButtonModel

This class makes use of two buttons to determine the speed. Its implementation can be found in the class `SpeedButtonModel`.

- `accelButtonIndex`
  This is the index used to identify the button for acceleration.
- `decelButtonIndex`
  This index points to a button used for deceleration.

SpeedMultiButtonModel

Several buttons are used in order to define the speed when using this model. It is implemented in the class `SpeedMultiButtonModel`.

- `accelButtonIndices`
  A list of button indices which are used for acceleration.
- `decelButtonIndices`
  A list of button indices which are used for deceleration.

SpeedAxisModel

An axis is used to define the motion speed when this model is used. The implementation of the model can be found in the class `SpeedAxisModel`.

- `axisIndex`
  This is the index of the axis to be used. Positive and negative values can accelerate and decelerate.
- `minThreshold`
  If the axis provides a value over the threshold the value used for speed manipulation otherwise it is not used.

SpeedDualAxisModel

The model is implemented in the class `SpeedDualAxisModel`. It makes use of two axes to manipulate the speed.

- `axis1Index`
  This is the index of the first axis to be used. Positive and negative values can accelerate and decelerate.
- `axis2Index`
  This is the index of the second axis to be used. Positive and negative values can accelerate and decelerate.

5.2 Summary

This chapter has given a brief overview on the setup of the navigation module. Three categories of models for translation, orientation and speed which are composed into a resulting transformation matrix were presented.
Since a variety of implementations exist in the different categories the configuration of the implementations was described in depth in the according sub-sections.

Chapter 6

Interaction Module

The configuration of the interaction module has to contain at least six different nodes, a seventh node is optional. Each of these nodes specifies a transition function for the interaction automaton of the *inVRs* framework. It is parsed by the `Interaction` class and successively by the models for the transition functions which are derived from the classes `StateActionModel` and `StateTransitionModel`. From the later class two child classes are derived again for further inheritance; the `ManipulationChangeModel` and `SelectionChangeModel`.
More detail on the inner workings of the interaction module can be found in the *Medieval Town Tutorial* and the *Programmers' Guide*.

- <stateActionModels>
  Models of this type are executed when the state machine is in one of the three interaction states.
  - <idleActionModel>
    This model is used for operations which are constantly executed during the idle state, when no object is selected or currently manipulated. Often the idle model is omitted.
  - <selectionActionModel>
    This model is used for the operations which are constantly executed during object selection. A common use of the model is highlighting the currently selected entity.
  - <manipulationActionModel>
    This model is used for the operations which are constantly executed during object manipulation. It works typically closely together with the chosen cursor model.
- <stateTransitionModels>
  In order to implement a change from one state to another state, the following transition modules have to be configured.
  - <selectionChangeModel>
    The selection change model is used to define the method of selection when the interaction automaton is still in an idle state.
  - <unselectionChangeModel>
    If the change from selection state back into the idle state takes place models of this type are used. They are typically the same models as the selection models applied in an inverse manner.
  - <manipulationConfirmationModel>
    In case the user wants to manipulate a selected object, the state has to change into manipulation which is realised by models of this type.
  - <manipulationTerminationModel>
    With this model it is possible to determine the behavior when switching from an active

entity manipulation back into the idle state. Typically the same model is used for manipulation termination as for manipulation confirmation.

```
<?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: exampleInteractionModule.xml

6.1 Interaction Models

A variety of interaction models are provided by the *inVRs* framework which can be combined to be used together as an interaction technique. All of these models have to be set in order as previously described to provide a valid technique.

6.1.1 Idle Action Models

The code of the idle action models is constantly executed during the idle state of the framework. They are derived from the class `StateActionModel`. Only one header is provided so far to represent the class `IdleActionModel`. Future implementation might use this state.

6.1.2 Selection Action Models

These models are used during the selection task of the interaction. Models of this class are derived from the superclass `StateActionModel`. As long as the user is in the selection state the configured model is executed.

HighlightSelectionActionModel

This model is implemented in the according class `HighlightSelectionActionModel` and performs a highlighting of the selected object. An additional semi-transparent object is displayed in a pulsating manner around the selected object.

- `modelType`
  The type of the model describes the object type which could be for example `3DS` or `VRML`.
- `modelPath`
  The path where the model for highlighting can be found.

6.1.3 Manipulation Action Models

These models are used during the manipulation task of the interaction. All of these models are derived from the superclass `StateActionModel`. The same mechanisms as for the selection action or the idle action models are used. They are constantly executed when the user is in the manipulation state.

VirtualHandManipulationActionModel

The model is used in the virtual hand interaction technique. It is implemented in the class `VirtualHandManipulationActionModel`. When this model is active a direct mapping from the sensor input on the `VirtualHandCursorModel` takes place.

HomerManipulationActionModel

This model is used to implement the HOMER interaction technique. The model is implemented in the class `HomerManipulationActionModel`. When the user leaves the selection state and enters the manipulation state the cursor is moved to the object which is to be manipulated. In order to implement a HOMER interaction technique the HOMER cursor model should be set up as well inside the user configuration.

- `usePickingOffset`
  If this attribute is set to true the cursor moves to the selected object at the first collision point. If it is set to false it moves to the centre of the object.

6.1.4 Selection Change and Unselection Change Models

These models are derived from the class `SelectionChangeModel`. It is possible to use them for both, the selection and as an inverse operation the unselection of objects.

RayCastSelectionChangeModel

This model is implemented in the class `RayCastSelectionChangeModel`. It is used for ray cast selection and unselection of entities. If a collision between ray originating from the cursor position and an entity takes place the automaton switches into selection state. If it is already in the selection state and no collision between the ray and the entity is detected the state is changed back to idle.

VirtualHandSelectionChangeModel

The model is implemented in the class `VirtualHandSelectionChangeModel`. It is probably the most traditional model when using VEs and tracking systems. Objects can be selected by placing the cursor in the inside of the model. The GoGo interaction technique for example can use a virtual hand model in case the GoGo cursor model is used inside the user configuration.

LimitRayCastSelectionChangeModel

The implementation of the model can be found in the class `LimitRayCastSelectionChangeModel`. It is very similar to the ray cast selection model. But instead of using an infinite ray a line of a defined length drawn from the cursor is used for entity selection and unselection.

- `rayDistanceThreshold`
  This attribute describes the length of the line used for selection.

6.1.5 Manipulation Confirmation and Termination Models

These models are implementations of the `ManipulationChangeModel`. This type of models can be used for the state change from selection to manipulation and vice versa.

ButtonPressStateTransitionModel

The model is implemented in the class `ButtonPressStateTransitionModel`. It is used for changing between interaction states once a button is pressed.

- `buttonIndex`
  The index of the button to be pressed in order to change between two states.

6.2 Summary

This chapter has briefly introduced the seven transition functions which have to be configured in the form of interaction models.
A variety of interaction techniques exist and are implemented for the *inVRs* framework. The available models used to compose an interaction technique were introduced and their specific configuration was described.

Chapter 7

Network Module

The setup of the network module is kept very slim. A node describes the ports where the application tries to communicate with, while another node is used for setting the local IP address. The configuration is parsed and evaluated by the class `Network`.
In the standard implementation of the *inVRs* network module all transformations passed down by the `TransformationDistributionModifier` are sent via UPD to all participants and all events distributed by the `EventManager` are transmitted via TCP.
Two nodes are used for configuring the network module.

- <ports>
  This node takes two arguments `TCP` describing the port for TCP communication and `UDP` describing the port for UDP communication.
- <localIP>
  The node can be configured with the attribute `value` which is to be set to the local IP address. In case two instances of an *inVRs* applications are launched on the same machine, this attribute has to be set different to localhost.

```
<?xml version="1.0"?>
<!DOCTYPE network SYSTEM "http://dtd.invrs.org/network_v1.0a4.dtd">
<network>
  <ports TCP="8081" UDP="8082" />
  <localIP value="140.178.104.34" />
</network>
```

Listing 7.1: exampleNetworkModule.xml

7.1 Summary

A very short chapter introducing the a brief setup of the *inVRs* standard network module configuration. The basic module allows only to set up the different ports were it communicates with other applications. When more than one instance are launched on a single machine the IP address has to be configured.
This configuration is to be significantly enhanced when other implementations of the network module are used.

Chapter 8

Outlook

This document has given a detailed overview on the configuration of the *inVRs* framework with its components. Initially conventions for naming, standard nodes and the reasons behind the file and path system were given. The general configuration has introduced the path setup as well as the basic component setup of the framework.
The system core with its sub-components was introduced in the second chapter. The user database hosting configuration for user representation and transformation of the user and the cursor was described. The world database used for storage and setup was explained with its components the tiles, the entities and the environments. The two managers for event and transformation management were briefly introduced, where the setup of the event manager cannot be configured yet.
The input interface configuration has illustrated how to integrate a variety of input devices and merge them together in an abstract controller. The subsequent chapter has given an idea on how to extend the output interface, which is unfortunately not configurable as well so far.
The configuration of the standard modules was described. The navigation module uses three different types of models in order to implement a flexible configuration of the navigation. Similar mechanisms are used by the interaction module to implement the interaction techniques by configuring transition functions. The configuration for the network module is kept very brief so far and designed for future extension.
Readers of this document should be now able to configure an *inVRs* application, setup the components and layout a VE.

8.1 Future Work

It is a document which will be constantly updated with additional information regarding interaction models and transition models as they are updated during the development of the framework. The main components which will have to be extended considering the configuration of the *inVRs* framework are:

- Event Manager
- Output Interface
- Network Module

To ease the creation of configuration files, an update on the *inVRs* editor is under development. It allows to graphically align and layout the VE and finally generate configurations for the world database.

8.2 Acknowledgments

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

Bibliography

[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.

[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.

[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.

[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.

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

[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.

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

List of Figures

Listings

Appendix

DTD

Argument Vector

```
<!ELEMENT arguments (arg*)>
<!ATTLIST arguments
  version (1.0a4) #REQUIRED>

<!ELEMENT arg EMPTY>
<!ATTLIST arg
  key CDATA #REQUIRED
  type CDATA #IMPLIED
  value CDATA #REQUIRED>
```

Listing 8.1: argumentVector_v1.0a4.dtd

Transformation

```
<!ELEMENT transformation (translation?,rotation?,scale?,scaleOrientation?)>

<!ELEMENT translation EMPTY>
<!ATTLIST translation
  x CDATA #REQUIRED
  y CDATA #REQUIRED
  z CDATA #REQUIRED>

<!ELEMENT rotation EMPTY>
<!ATTLIST rotation
  x CDATA #REQUIRED
  y CDATA #REQUIRED
  z CDATA #REQUIRED
  angleDeg CDATA #IMPLIED
  angleRad CDATA #IMPLIED>

<!ELEMENT scale EMPTY>
<!ATTLIST scale
  x CDATA #REQUIRED
  y CDATA #REQUIRED
  z CDATA #REQUIRED>

<!ELEMENT scaleOrientation EMPTY>
<!ATTLIST scaleOrientation
  x CDATA #REQUIRED
  y CDATA #REQUIRED
  z CDATA #REQUIRED
  angleDeg CDATA #IMPLIED
  angleRad CDATA #IMPLIED>
```

Listing 8.2: transformation_v1.0a4.dtd

Representation

```
<!ENTITY % transformation SYSTEM "transformation_v1.0a4.dtd" >
%transformation;

<!ELEMENT representation (( file | reference ), transformation?)>
<!ATTLIST representation
  copy CDATA #IMPLIED>

<!ELEMENT file EMPTY>
<!ATTLIST file
  type CDATA #REQUIRED
  name CDATA #REQUIRED>

<!ELEMENT reference EMPTY>
<!ATTLIST reference
  sourceId CDATA #REQUIRED>
```

Listing 8.3: representation_v1.0a4.dtd

General Config

```
<!ELEMENT generalConfig (general, paths)>
<!ATTLIST generalConfig
  version (1.0a4) #REQUIRED>

<!ELEMENT general ANY>

<!ELEMENT paths (root, path*)>

<!ELEMENT root EMPTY>
<!ATTLIST root
  directory CDATA #REQUIRED>

<!ELEMENT path EMPTY>
<!ATTLIST path
  name CDATA #REQUIRED
  directory CDATA #REQUIRED>
```

Listing 8.4: generalConfig_v1.0a4.dtd

Modules

```
<!ELEMENT modules (module*)>
<!ATTLIST modules
  version (1.0a4) #REQUIRED>

<!ELEMENT module EMPTY>
<!ATTLIST module
  name CDATA #REQUIRED
  configFile CDATA #REQUIRED
  libraryName CDATA #IMPLIED>
```

Listing 8.5: modules_v1.0a4.dtd

System Core

```
<!ELEMENT systemCore (worldDatabase, userDatabase, transformationManager,
    eventManager?)>
<!ATTLIST systemCore
  version (1.0a4) #REQUIRED>
```

```
<!ELEMENT worldDatabase EMPTY>
<!ATTLIST worldDatabase
  configFile CDATA #REQUIRED>

<!ELEMENT userDatabase EMPTY>
<!ATTLIST userDatabase
  configFile CDATA #REQUIRED>

<!ELEMENT transformationManager EMPTY>
<!ATTLIST transformationManager
  configFile CDATA #REQUIRED>

<!ELEMENT eventManager EMPTY>
<!ATTLIST eventManager
  configFile CDATA #REQUIRED>
```

Listing 8.6: systemCore_v1.0a4.dtd

User Database

```
<!ELEMENT userDatabase (avatar?, cursorRepresentation?, cursorTransformationModel?,
     userTransformationModel?)>
<!ATTLIST userDatabase
  version (1.0a4) #REQUIRED>

<!ELEMENT avatar EMPTY>
<!ATTLIST avatar
  configFile CDATA #REQUIRED>

<!ELEMENT cursorRepresentation EMPTY>
<!ATTLIST cursorRepresentation
  configFile CDATA #REQUIRED>

<!ELEMENT cursorTransformationModel EMPTY>
<!ATTLIST cursorTransformationModel
  configFile CDATA #REQUIRED>

<!ELEMENT userTransformationModel EMPTY>
<!ATTLIST userTransformationModel
  configFile CDATA #REQUIRED>
```

Listing 8.7: userDatabase_v1.0a4.dtd

Simple Avatar

```
<!ENTITY % representation SYSTEM "representation_v1.0a4.dtd" >
%representation;

<!ELEMENT SimpleAvatar (name?, representation)>
<!ATTLIST SimpleAvatar
  version (1.0a4) #REQUIRED>

<!ELEMENT name EMPTY>
<!ATTLIST name
  value CDATA #REQUIRED>
```

Listing 8.8: simpleAvatar_v1.0a4.dtd

Avatara Avatar

```
<!ENTITY % representation SYSTEM "representation_v1.0a4.dtd" >
%representation;

<!ELEMENT AvataraAvatar (name?, representation, texture?, animations?)>
<!ATTLIST AvataraAvatar
  version (1.0a4) #REQUIRED>

<!ELEMENT name EMPTY>
<!ATTLIST name
  value CDATA #REQUIRED>

<!ELEMENT texture EMPTY>
<!ATTLIST texture
  file CDATA #REQUIRED>

<!ELEMENT animations (animation*)>
<!ATTLIST animations
  smooth CDATA #IMPLIED
  speed CDATA #IMPLIED
  default CDATA #IMPLIED>

<!ELEMENT animation EMPTY>
<!ATTLIST animation
  name CDATA #REQUIRED
  file CDATA #REQUIRED>
```

Listing 8.9: avataraAvatar_v1.0a4.dtd

Cursor Transformation Model

```
<!ELEMENT cursorTransformationModel (model)>
<!ATTLIST cursorTransformationModel
  version (1.0a4) #REQUIRED>

<!ELEMENT model (arguments?)>
<!ATTLIST model
  name CDATA #REQUIRED>
```

Listing 8.10: cursorTransformationModel_v1.0a4.dtd

Simple Cursor Representation

```
<!ENTITY % representation SYSTEM "representation_v1.0a4.dtd" >
%representation;

<!ELEMENT simpleCursorRepresentation (representation)>
<!ATTLIST simpleCursorRepresentation
  version (1.0a4) #REQUIRED>
```

Listing 8.11: simpleCursorRepresentation_v1.0a4.dtd

Interaction Cursor Representation

```
<!ENTITY % representation SYSTEM "representation_v1.0a4.dtd" >
%representation;

<!ELEMENT interactionCursorRepresentation (idleModel, selectionModel?,
    manipulationModel?)>
<!ATTLIST interactionCursorRepresentation
  version (1.0a4) #REQUIRED>
```

```
<!ELEMENT idleModel (representation)>
<!ATTLIST idleModel>

<!ELEMENT selectionModel (representation)>
<!ATTLIST selectionModel>

<!ELEMENT manipulationModel (representation)>
<!ATTLIST manipulationModel>
```

Listing 8.12: interactionCursorRepresentation_v1.0a4.dtd

User Transformation Model

```
<!ELEMENT userTransformationModel (model)>
<!ATTLIST userTransformationModel
  version (1.0a4) #REQUIRED>

<!ELEMENT model (arguments?)>
<!ATTLIST model
  name CDATA #REQUIRED>
```

Listing 8.13: userTransformationModel_v1.0a4.dtd

World Database

```
<!ELEMENT worldDatabase (entityTypes*, tiles*, environmentLayout?)>
<!ATTLIST worldDatabase
  version (1.0a4) #REQUIRED>

<!ELEMENT entityTypes EMPTY>
<!ATTLIST entityTypes
  configFile CDATA #REQUIRED>

<!ELEMENT tiles EMPTY>
<!ATTLIST tiles
  configFile CDATA #REQUIRED>

<!ELEMENT environmentLayout EMPTY>
<!ATTLIST environmentLayout
  configFile CDATA #REQUIRED>
```

Listing 8.14: worldDatabase_v1.0a4.dtd

Entity Types

```
<!ENTITY % representation SYSTEM "representation_v1.0a4.dtd" >
%representation;

<!ELEMENT entityTypes (entityType*)>
<!ATTLIST entityTypes
      version (1.0a4) #REQUIRED>

<!ELEMENT entityType (representation, implementationClass?)>
<!ATTLIST entityType
      typeId CDATA #REQUIRED
      name CDATA #REQUIRED
      fixed CDATA #IMPLIED
      implementationClass CDATA #IMPLIED>

<!ELEMENT implementationClass ANY>
```

Listing 8.15: entityTypes_v1.0a4.dtd

Tiles

```
<!ENTITY % representation SYSTEM "representation_v1.0a4.dtd" >
%representation;

<!ELEMENT tiles (tile*)>
<!ATTLIST tiles
  version (1.0a4) #REQUIRED>

<!ELEMENT tile (tileProperties, representation)>
<!ATTLIST tile
  id CDATA #REQUIRED
  name CDATA #REQUIRED>

<!ELEMENT tileProperties (size, adjustment)>

<!ELEMENT size (#PCDATA)>
<!ATTLIST size
  xSize CDATA #REQUIRED
  zSize CDATA #REQUIRED>

<!ELEMENT adjustment (#PCDATA)>
<!ATTLIST adjustment
  height CDATA #REQUIRED
  yRotation CDATA #REQUIRED>
```

Listing 8.16: tiles_v1.0a4.dtd

Environment Layout

```
<!ELEMENT environmentLayout (tileGrid, environment*)>
<!ATTLIST environmentLayout
  version (1.0a4) #REQUIRED>

<!ELEMENT tileGrid EMPTY>
<!ATTLIST tileGrid
  xSpacing CDATA #REQUIRED
  zSpacing CDATA #REQUIRED>

<!ELEMENT environment EMPTY>
<!ATTLIST environment
  id CDATA #REQUIRED
  configFile CDATA #REQUIRED
  xLoc CDATA #REQUIRED
  zLoc CDATA #REQUIRED>
```

Listing 8.17: environmentLayout_v1.0a4.dtd

Environment

```
<!ENTITY % transformation SYSTEM "transformation_v1.0a4.dtd" >
%transformation;

<!ELEMENT environment (tileMap, entryPoint*, entity*)>
<!ATTLIST environment
  version (1.0a4) #REQUIRED>
```

```
<!ELEMENT tileMap (#PCDATA)>
<!ATTLIST tileMap
  xDimension CDATA #REQUIRED
  zDimension CDATA #REQUIRED>

<!ELEMENT entryPoint (#PCDATA)>
<!ATTLIST entryPoint
  xPos CDATA #REQUIRED
  yPos CDATA #REQUIRED
  zPos CDATA #REQUIRED
  xDir CDATA #REQUIRED
  yDir CDATA #REQUIRED
  zDir CDATA #REQUIRED>

<!ELEMENT entity (transformation)>
<!ATTLIST entity
  id CDATA #REQUIRED
  typeId CDATA #REQUIRED>
```

Listing 8.18: environment_v1.0a4.dtd

Event Manager

```
<!ELEMENT eventManager (#PCDATA)>
<!ATTLIST eventManager
  version (1.0a4) #REQUIRED>
```

Listing 8.19: eventManager_v1.0a4.dtd

Transformation Manager

```
<!ENTITY % arguments SYSTEM "argumentVector_v1.0a4.dtd" >
%arguments;

<!ELEMENT transformationManager (logFile?, mergerList?, pipeList?)>
<!ATTLIST transformationManager
  version (1.0a4) #REQUIRED>

<!ELEMENT logFile EMPTY>
<!ATTLIST logFile
  name CDATA #REQUIRED>

<!ELEMENT mergerList (merger*)>
<!ATTLIST mergerList>

<!ELEMENT merger (inputPipe+, outputPipe+, arguments?)>
<!ATTLIST merger
  type CDATA #REQUIRED
  id CDATA #REQUIRED>

<!ELEMENT inputPipe EMPTY>
<!ATTLIST inputPipe
  srcComponent CDATA #IMPLIED
  srcComponentName CDATA #IMPLIED
  dstComponent CDATA #IMPLIED
  dstComponentName CDATA #IMPLIED
  pipeType CDATA #REQUIRED
  objectClass CDATA #REQUIRED
  objectType CDATA #REQUIRED
  objectId CDATA #REQUIRED
  fromNetwork CDATA #REQUIRED>

<!ELEMENT outputPipe EMPTY>
```

```
<!ATTLIST outputPipe
  srcComponent CDATA #IMPLIED
  srcComponentName CDATA #IMPLIED
  dstComponent CDATA #IMPLIED
  dstComponentName CDATA #IMPLIED
  pipeType CDATA #REQUIRED
  objectClass CDATA #REQUIRED
  objectType CDATA #REQUIRED
  objectId CDATA #REQUIRED
  fromNetwork CDATA #REQUIRED>

<!ELEMENT pipeList (pipe*)>
<!ATTLIST pipeList>

<!ELEMENT pipe (modifier*)>
<!ATTLIST pipe
  srcComponent CDATA #IMPLIED
  srcComponentName CDATA #IMPLIED
  dstComponent CDATA #IMPLIED
  dstComponentName CDATA #IMPLIED
  pipeType CDATA #REQUIRED
  objectClass CDATA #REQUIRED
  objectType CDATA #REQUIRED
  objectId CDATA #REQUIRED
  fromNetwork CDATA #REQUIRED>

<!ELEMENT modifier (arguments?)>
<!ATTLIST modifier
  type CDATA #REQUIRED>
```

Listing 8.20: transformationManager_v1.0a4.dtd

Input Interface

```
<!ELEMENT inputInterface (module*)>
<!ATTLIST inputInterface
  version (1.0a4) #REQUIRED>

<!ELEMENT module EMPTY>
<!ATTLIST module
  name CDATA #REQUIRED
  configFile CDATA #REQUIRED
  libraryName CDATA #IMPLIED>
```

Listing 8.21: inputInterface_v1.0a4.dtd

Controller Manager

```
<!ENTITY % transformation SYSTEM "transformation_v1.0a4.dtd" >
%transformation;
<!ENTITY % arguments SYSTEM "argumentVector_v1.0a4.dtd" >
%arguments;

<!ELEMENT controllerManager (controller)>
<!ATTLIST controllerManager
  version (1.0a4) #REQUIRED>

<!ELEMENT controller (device*)>
<!ATTLIST controller
  buttons CDATA #REQUIRED
  axes CDATA #REQUIRED
  sensors CDATA #REQUIRED>
```

```
<!ELEMENT device (arguments?, button*, axis*, sensor*)>
<!ATTLIST device
  type CDATA #REQUIRED>

<!ELEMENT button (buttonCorrection?)>
<!ATTLIST button
  deviceIndex CDATA #REQUIRED
  controllerIndex CDATA #REQUIRED>

<!ELEMENT buttonCorrection EMPTY>
<!ATTLIST buttonCorrection
  invert CDATA #REQUIRED>

<!ELEMENT axis ((axisCorrection|axisValues)?)>
<!ATTLIST axis
  deviceIndex CDATA #REQUIRED
  controllerIndex CDATA #REQUIRED>

<!ELEMENT axisCorrection EMPTY>
<!ATTLIST axisCorrection
  offset CDATA #REQUIRED
  scale CDATA #REQUIRED>

<!ELEMENT axisValues EMPTY>
<!ATTLIST axisValues
  minValue CDATA #REQUIRED
  maxValue CDATA #REQUIRED>

<!ELEMENT sensor (coordinateSystemCorrection?, positionCorrection?,
    orientationCorrection?)>
<!ATTLIST sensor
  deviceIndex CDATA #REQUIRED
  controllerIndex CDATA #REQUIRED>

<!ELEMENT coordinateSystemCorrection (translation?, rotation?, scale?)>
<!ATTLIST coordinateSystemCorrection>

<!ELEMENT positionCorrection (translation?, scale?)>
<!ATTLIST positionCorrection>

<!ELEMENT orientationCorrection (rotation?)>
<!ATTLIST orientationCorrection>
```

Listing 8.22: controllerManager_v1.0a4.dtd

Output Interface

```
<!ELEMENT outputInterface (module*)>
<!ATTLIST outputInterface
  version (1.0a4) #REQUIRED>

<!ELEMENT module EMPTY>
<!ATTLIST module
  name CDATA #REQUIRED
  configFile CDATA #REQUIRED
  libraryName CDATA #IMPLIED>
```

Listing 8.23: outputInterface_v1.0a4.dtd

Navigation Module

```
<!ENTITY % arguments SYSTEM "argumentVector_v1.0a4.dtd" >
%arguments;
```

```
<!ELEMENT navigation (translationModel?, orientationModel?, speedModel?)>
<!ATTLIST navigation
  version (1.0a4) #REQUIRED>

<!ELEMENT translationModel (arguments?)>
<!ATTLIST translationModel
  type CDATA #REQUIRED>

<!ELEMENT orientationModel (arguments?)>
<!ATTLIST orientationModel
  type CDATA #REQUIRED
  angle CDATA #REQUIRED>

<!ELEMENT speedModel (arguments?)>
<!ATTLIST speedModel
  type CDATA #REQUIRED
  speed CDATA #REQUIRED>
```

Listing 8.24: navigation_v1.0a4.dtd

Interaction Module

```
<!ENTITY % arguments SYSTEM "argumentVector_v1.0a4.dtd" >
%arguments;

<!ELEMENT interaction (stateActionModels, stateTransitionModels)>
<!ATTLIST interaction
  version (1.0a4) #REQUIRED>

<!ELEMENT stateActionModels (idleActionModel?, selectionActionModel?,
    manipulationActionModel?)>
<!ATTLIST stateActionModels>

<!ELEMENT idleActionModel (arguments?)>
<!ATTLIST idleActionModel
  type CDATA #REQUIRED>

<!ELEMENT selectionActionModel (arguments?)>
<!ATTLIST selectionActionModel
  type CDATA #REQUIRED>

<!ELEMENT manipulationActionModel (arguments?)>
<!ATTLIST manipulationActionModel
  type CDATA #REQUIRED>

<!ELEMENT stateTransitionModels (selectionChangeModel?, unselectionChangeModel?,
    manipulationConfirmationModel?, manipulationTerminationModel?)>
<!ATTLIST stateTransitionModels>

<!ELEMENT selectionChangeModel (arguments?)>
<!ATTLIST selectionChangeModel
  type CDATA #REQUIRED>

<!ELEMENT unselectionChangeModel (arguments?)>
<!ATTLIST unselectionChangeModel
  type CDATA #REQUIRED>

<!ELEMENT manipulationConfirmationModel (arguments?)>
<!ATTLIST manipulationConfirmationModel
  type CDATA #REQUIRED>

<!ELEMENT manipulationTerminationModel (arguments?)>
<!ATTLIST manipulationTerminationModel
  type CDATA #REQUIRED>
```

Listing 8.25: interaction_v1.0a4.dtd

Network Module

```
<!ELEMENT network (ports, localIP?)>
<!ATTLIST network
  version (1.0a4) #REQUIRED>

<!ELEMENT ports EMPTY>
<!ATTLIST ports
  TCP CDATA #REQUIRED
  UDP CDATA #REQUIRED>

<!ELEMENT localIP EMPTY>
<!ATTLIST localIP
  value CDATA #REQUIRED>
```

Listing 8.26: network_v1.0a4.dtd