EC DG-INFSO
IST Project IST-2001-34024

Specification of the MR framework

Deliverable Number:	**D5.2**
Revision:	**1.0**
WPs related to the Deliverable:	**WP 5**
Type (Internal, Restricted, Public):	**Public**
Nature of the Deliverable:	**Report**
Contractual Date of Delivery:	**30/09/2002**
Actual date of Delivery:	**25/06/2002**
Authors(s):	**FHH: Wilhelm Burger** **Michael Haller** **Jürgen Zauner** **FAW: Thomas Luckeneder** **Werner Hartmann**

Executive Summary

This delivery contains the specification of the MR framework as detailed as possible now. The further development of the requirements of the authoring framework and the demonstrator applications may enforce minor changes of this specification. But the foundations of the framework are flexible enough to allow such changes.

The first part of the report contains a description of the architecture of the AMIRE framework and the definition of two major building parts of the framework, components and gems. It also describes the structured properties, a generic mechanism that allows defining the interfaces of components. Those structured properties build also the foundation for the framework's component communication system.

The next section describes the development environment that was chosen for the implementation of the AMIRE frame work. The programming language, the compiler and the version control system are selected within this section.

After this, some widespread and commonly used component systems like CORBA, COM and EJB are described and it is evaluated, whether they can be used for the AMIRE framework or not. The evaluation stated that none of the existing component systems can be used as the core of the AMIRE framework, but it is possible to build AMIRE components that wrap components of those general purpose component systems.

The subsequent section introduces the so called main library gems. Those main library gems are software modules that are tightly coupled with the framework. Thus, the selection of those gems is important for the framework. A set of requirements that those main library gems have to fulfil was established. To provide profound information about candidate implementations for those main library gems, and to find out how they comply with the requirements, two feasibility studies where carried out. The results of those studies are described in the next section.

The final part of the report contains coding guidelines that the source code of AMIRE must comply with. The reasons for these guidelines are to foster the developers of AMIRE to write well readable code and to get a uniform look of the source code. This should ease the cooperation of the distributed developers of AMIRE, and raise the quality of the written code.

Distribution List

Internal to the consortium

●	FHH	
●	FAW	
●	LABEIN	
●	c-Lab	
●	AGC	
●	TC	
●	HUT	
●		
●		
●		

External to the consortium

●	Project Officer	Eric Badiqué

Document Change Log

Revision number	Issue date	Sections affected	Comments

Table of contents

EXECUTIVE SUMMARY 2

1. INTRODUCTION 8

1.1. Purpose 8

1.2. General objectives 8

1.3. Overview 8

2. ARCHITECTURE 8

2.1. Overview 8

2.2. Layers 8

2.3. Definition of gems 8

2.4. Definition of components 8

2.5. Gems versus components 8

3. GEM CLASSIFICATION 8

3.1. Low-level gems 8

3.2. High-level gems 8

3.3. Main library gems 8

3.4. Inter-gem communication 8

3.4.1. Direct communication 8

3.4.2. Listener communication 8

4. COMPONENT CLASSIFICATION 8

4.1. Wrapper components 8

4.2. Customized components 8

4.3. Composed components 8

5. PROPERTIES 8

5.1. Typing 8
5.1.1. Reference property type 8
5.1.2. Simple base property types 8
5.1.3. Structured base property type 8
5.1.4. Vector property type 8

5.2. Property type manager 8

5.3. Persistence 8
5.3.1. Export 8
5.3.2. Import 8

5.4. Specification 8
5.4.1. Property class 8
5.4.2. ReferenceProperty class 8
5.4.3. PersistentProperty class 8
5.4.4. TBaseProperty template class 8
5.4.5. TBaseVectorProperty template class 8
5.4.6. StructProperty class 8
5.4.7. StructVectorProperty class 8
5.4.8. PropertyType class 8
5.4.9. VectorPropertyType class 8
5.4.10. StructPropertyType class 8
5.4.11. PropertyTypeManager class 8
5.4.12. PropertyReader class 8
5.4.13. PropertyWriter class 8

COMPONENTS 8

5.5. Inter-object communication concepts 8
5.5.1. Signals and slots 8
5.5.2. Beans and properties 8
5.5.3. State oriented listeners 8

5.6. Inter-Component communication 8
5.6.1. In- and out-slots 8
5.6.2. Typing and connectability 8
5.6.3. Connections 8

5.7. Configuration 8

5.8. Composed components 8

5.9. Component manager 8

5.10. Persistence 8

5.10.1. Export 8
5.10.2. Import 8

5.11. Specification 8
5.11.1. Component class....8
5.11.2. ComponentState class....**Fehler! Textmarke nicht definiert.**
5.11.3. Connection class....**Fehler! Textmarke nicht definiert.**
5.11.4. ConnectionFilter class ....**Fehler! Textmarke nicht definiert.**
5.11.5. ComposedComponent class....8
5.11.6. ComponentManager class 8
5.11.7. ComponentManagerReader class 8
5.11.8. ComponentManagerWriter class 8

6. DEVELOPMENT ENVIRONMENT 8

7. COMPONENT TECHNOLOGIES 8

7.1. CORBA 8

7.2. COM and DCOM 8

7.3. EJB and JavaBeans 8

8. X3D....8

9. XML BASED PERSISTENCE....8

9.1. Properties 8

9.2. Component manager 8

10. MAIN LIBRARY GEMS 8

10.1. Main Library Gem Groups and Categories 8
10.1.1. Loader Group 8
10.1.2. Positioning System Group....8

10.2. Main Library Gem Requirements....8

10.3. Loader Group Gem Candidates 8
10.3.1. Principles of Scene Graphs....8
10.3.2. OpenSG 8
10.3.3. Open Scene Graph....8

10.4. Tracking Group Gem Candidates....8
10.4.1. Principles of optical tracking....8

10.4.2. ARToolkit 8
10.4.3. TRIP 8
10.4.4. OpenCV 8

11. FEASIBILITY STUDIES 8

11.1. Tasks 8

11.2. Integration of ARToolkit and OpenSG 8
11.2.1. Solution 8
11.2.2. Evaluation 8

11.3. Integration of ARToolkit and OpenSceneGraph 8
11.3.1. Solution 8
11.3.2. Evaluation 8

11.4. Conclusion 8

12. C++ PROGRAMMING GUIDELINES 8

12.1. Language independent rules 8
12.1.1. Unary operators 8
12.1.2. Binary operator 8
12.1.3. Brackets 8
12.1.4. Statements and statement delimiters 8
12.1.5. Indentation 8
12.1.6. Font 8
12.1.7. Line length 8

12.2. Control structures 8
12.2.1. if 8
12.2.2. while 8
12.2.3. for 8
12.2.4. do while 8
12.2.5. switch 8

12.3. Names 8
12.3.1. Language 8
12.3.2. Methods, namespaces, enumerations, variables and attributes 8
12.3.3. Classes and types 8
12.3.4. Defined constant values and macros 8

12.4. Classes 8

12.5. Namespaces 8

12.6. Header and Source files 8

12.6.1. Prevent re-including 8
12.6.2. Head information....8

12.7. Documentation 8

APPENDIX A –REFERENCES....8

APPENDIX B – GLOSSARY 8

Table of figures

Figure 1: Layers of the AMIRE framework 8
Figure 2: Listener concept 8
Figure 3: Overview of the property types 8
Figure 4: Property type manager 8
Figure 5: Exporting a property 8
Figure 6: Importing a property 8
Figure 7: Property value class hierarchy 8
Figure 8: Basic property classes 8
Figure 9: Basic vector property classes 8
Figure 10: Property type class hierarchy 8
Figure 11: Relations between the class PropertyTypeManager and the class PropertyTypes 8
Figure 12: Detailed relations between property types and property type manager 8
Figure 13: Component interfaces 8
Figure 14: Component connections 8
Figure 15: Composed component 8
Figure 16: Component manager 8
Figure 17: Export of components and connections 8
Figure 18: Import of components and conncetions 8
Figure 19: Client access the object implementation via the ORB 8
Figure 20: Interfaces of the ORB 8
Figure 21: Implementation of CORBA objects 8
Figure 22: Generic wrapper components over CORBA objects 8
Figure 23: Example of the CORBA wrapping process 8
Figure 24: High level scene management with scene graphs 8
Figure 25: Scene Graph structure 8
Figure 26: Basic Structure of OpenSG 8
Figure 27: `NodeCores` of OpenSG 8
Figure 28: `NodeCore` Sharing 8
Figure 29: Image processing chain of optical tracking systems 8
Figure 30 : Pinhole Camera 8
Figure 31 : Transformation from camera frame coordinates to pixel coordinates 8
Figure 32 : Extrinsic camera parameters 8
Figure 33 : ARToolkit Marker 8
Figure 34 : TRIP marker 8

Table of tables

1. Introduction

1.1. Purpose

The framework is a central part of the AMIRE project. It supports the gem developer by providing easy and convenient ways to integrate gems; it supports the component builder by enabling the building of components out of gems and by providing tools to define component interfaces. The framework enables the component builder to establish connections between components and to build complex components out of simple ones. It builds the foundation of the authoring framework by providing basic methods to determine whether components are connectable or not.

1.2. General objectives

AMIRE's goal and objective is not to develop new base technologies for mixed reality applications, but to adopt existing solutions and provide them in a uniform way. Accordingly, the scientific objectives are focused in the area of component based modeling, Mixed Reality and human computer interaction especially in authoring environments.

The AMIRE architecture includes the following packages:

- MR gems,
- MR components, and
- MR framework

Similar to existing gems collections (game programming gems, graphic gems) AMIRE will produce a MR gem collection containing efficient solutions to individual mixed reality problems. A gem could be an object recognition library, a 3ds object loader, a solution for media generation (2D, 3D, audio, text) or a high level animation. In our case we concentrate on MR based gems. The gem collection will be used in the AMIRE framework but can also be used in other mixed reality projects. There will be ensured that useful existing MR technology software units will be available in a well-defined way, so that they can be used several times in MR applications as well as in the MR authoring tools. There will be MR gems for using augmented reality technologies (e.g. MR gems for identifying markers in a video image) as well as for using rich media (e.g. MR gems for handling and mapping video in a computer generated 3D scene). Typically, AMIRE gems can be reused in many different MR based applications. For example, a "magical lens"-gem that allows seeing through walls of a museum showing the location of a special art piece to the visitor may also be reused to explain the inner parts of the machine that cannot be physically opened by the trainee. The MR gems in turn can be used to build application specific MR components as well as an MR framework that defines how MR components can communicate with each other and can be integrated in an MR application.

MR components represent solutions for particular specific problems and they typically combine and extend MR gems towards advanced high-level features of an MR application. MR components feature a unified interface that easily allows configuring and combining them via the MR authoring tools using a suitable component model. For example, an object recognition gem, a path animation gem and a 3D object loader gem may be combined into a "magical lens"-component that illustrates the inner details of a real machine by providing a suitable animation of virtual machine parts.

So the primary goal of the MR framework package is to define the core framework. The main requirements of this framework are:

- Support for the management of the base components and gems for different types of use. The modules of this core system include a Gems Repository and a Component Management Unit.

For the gems we do not plan a Gems Management Unit, because they "only" represent an efficient implementation of different partial solutions.

- The integration of the Object Recognition Unit (e.g. ARToolkit).
- The integration of the Graphics Library (OpenGL, Direct3D, OpenSG…)
- The Component Loader unit allows the integration of the different objects into the system.
- All used objects and object types that are available in the system and stored in a database are registered and stored in the Repository Unit.
- The Message Handler identifies the components that are being addressed, delivers the message and maps it to the corresponding function calls.
- And finally, the Execution Unit handles the execution of all actions that appear in the system.

Moreover the framework has to perform the following criteria:

- Easy to use for both users: the expert authoring user who uses the gems and wants an easy integration of complex gems in their MR application and the end users who do not want to be confronted with the huge technical complexity of a MR application.
- Flexibility and scalability

1.3. Overview

This section gives a short overview of the document and its structure. The document starts with a small introduction providing the reader with the basic ideas, objectives and purposes of the MR framework.

After the introduction the architecture of the MR framework is introduced to the reader. It gives him an overview of the layers inside the MR framework and the differences between gems and components.

In the next chapter the reader gets more detailed information about main library gems, their classification and support in the MR framework.

Like the classification chapter about the main library gems the component classification chapter identifies the different kinds of components. This is important for the understanding of the component developer requirements, which have to be supported by the MR framework.

After the classification the property concept is introduced to the reader. This is a real essential concept of the MR framework, because it builds the foundation for the component concept and the component communication system. So it is important to understand this concept to be able to understand the components of the MR framework.

In the last chapter the details of the recommended component concept of the MR framework are described. It begins with basic ideas of other tools and programming languages. Based on this collection of basic ideas and all requirements of component developers the component concept is build up. So the reader should be able to understand the concept and its history.

2. Architecture

2.1. Overview

A gem collection is the base of all applications. Without the gems there is no functionality and no application. But a set of gems will not support you to build an application easily and fast. Maintenance is also a problem when different programmers create gems and there exists nothing holding them all together. So the job of gem developers is to develop or find new gems. The framework developer has to deliver conventions and plug-in interfaces for these gems. This means he has to classify gems with common functionality. Each of these gems must have the same look-and-feel. When it is possible he must find a plug-in interface for this class of gems. For example a plug-in interface for an image loader would create an abstraction of the implementation. The gem developer has to create image loaders for each image format and register them at the plug-in manager. The problem with gem plug-ins is that all gems are potential different structured and have different behaviors. This makes it impossible to use one plug-in interface and manager. The framework must provide plug-in mechanisms for all gem classes allowing such a mechanism. Gem classes that can't be abstracted by plug-ins must be abstracted by the use of object oriented design. Most important for an easy to use framework is the use of conventions. Such conventions will prescribe the design of plug-ins, class names and methods. For example loaders have in common that they will support specific file formats, load supported files and deliver objects containing the content of the files. Due to the different structure of the objects it makes no sense to design a basic loader class. A convention describing the basic design and behavior of loaders is much better.
Traditionally programmers develop applications. The programmer applies the gems and concatenates them to more complex constructs. This will finally lead to a construct representing the application. For an easy concatenation of gems the framework provides a low-level communication mechanism, which is more abstract and extensible than direct method calls. This communication mechanism is based on the popular observer pattern [GamHel95]. It allows the observation of model changes.
In contrast to the traditionally approach of developing applications by a programmer there exists the development by an authoring tool. This means an expert in the application field who is not familiar with programming will create the application by using a tool. This tool has to abstracts the programming processes. So the framework also must support the developer of the authoring tool by defining the interfaces and basic behavior of components. Therefore the framework provides a high-level communication mechanism. It allows the concatenation of generic components. Further the framework supports the generic configuration of components. So the user of the authoring tool can create the application by creating instances of components and concatenate them. After the creation of a component network the framework provides the ability to make the network persistent.

2.2. Layers

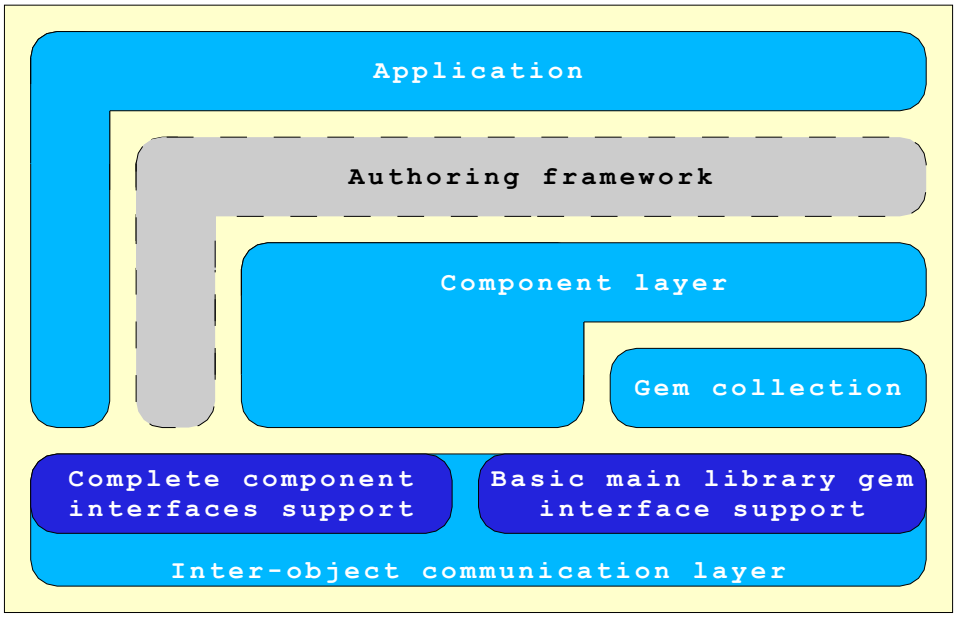

Figure 1: Layers of the AMIRE framework

Figure 1 describes the different layers of AMIRE's architecture. On the base layer we have the inter-object communication layer which allows the communication of both, the components and the gems. This communication layer provides generic interfaces and mechanisms for component management, communication and authoring. It also includes plug-in interfaces for easy gem extension. Further a gem communication mechanism based on the listener concept is provided. Based on the gem collection we have the component layer. A component includes one or more different gems. Finally, the authoring tool and the application tool work with all the underlying layers.
The MR framework is the glue of the AMIRE project that integrates both, the MR gems and the MR components. The main requirements of the framework are:

- The integration of the Graphics Library (actually we tend to support OpenGL and OpenSG or OpenSceneGraph)
- The integration of an object recognition unit (e.g. ARToolKit)
- The integration of different tracking systems (e.g. Polhemus)
- The component interface; this includes a component loader unit that allows the integration of different well-defined objects into the system.

A message handling architecture, in which the message handler identifies the components that are being addressed, delivers the message and maps it to the corresponding function calls.

2.3. Definition of gems

Gems provide the functionality required by the developer for the implementation of the application. Due to this they are essential for developing applications. The application is built of different gems. These gems are connected by the developer into a gem network, which represents the application. Each gem handles a special requirement or functionality of the application. A gem for example is

responsible for loading an image. Another gem provides the visibility calculation of 3D geometry. Gems are often built out of other gems, which handle smaller problems.
The developer must reuse such gems in his/her application. This ensures the quality of the application and makes the maintenance of the application much easier. The developer has to specify the problem he/she currently tries to solve. After the specification he tries to find an existing gem already handling the problem. The gem may be a code snippet occurring in a book or another source code. It also can be a function in a library. Or it even can be theoretical work, which is part of papers or books. After a successful search, the developer has to integrate these gems into his/her application. This means he/she has to clarify the license or copyright issues and take over the source, use the library function or implement the theory.

2.4. Definition of components

Like a complex gem a component is a gem network, handling a specific functionality. Nevertheless it provides not enough functionality to be an application. But it has generic interfaces that are common for all current and future components. These interfaces handle:

- the instantiation of components by a name description of the desired component.
- the configuration of component instances by using a generic description of structured data.
- the persistence of components by reusing the configuration mechanism and structured data description.
- the concatenation of components into a component network by using name descriptions of the action to perform and the transferred structured data.

2.5. Gems versus components

In contrast to components we cannot support generic and common interfaces for all kind of gems. Only a part of the main library gems can be supported by a small set of reusable communication and plug-in interfaces. They have no common configuration mechanism and they have not to be persistent. The developer of a gem-based application connects gems by using direct method calls. The developer of a component-based application connects the component by using the generic interfaces. So the connections between gems are created at the compile time and the connections between components at the runtime. Due to the generic interfaces of components and the runtime connections it is possible to create the connections in an authoring tool.

3. Gem classification

3.1. Low-level gems

Low-level gems are gems that are provided by other frameworks or libraries. A sound library as OpenAL for example is a low-level gem providing the functionality of 3D sound sources. Another low-level gem for example is a commercial encryption algorithm provided by a company. Basically most of the low-level gems have in common that there is no option to immediately change or manipulate the sources of them. This means the maintenance work has to be done by the provider of the library.

3.2. High-level gems

High-level gems are gems, which are available as source code and could be integrated into the framework. This can be source snippets or gem that are built out of other gems. For example you have a geometry loader as a gem and a gem that is able to clip geometry by several planes. With these two gems you are able to create a gem that load the geometry and the clipping planes and creates the new geometry that is inside of the clipping planes. The maintenance work for high-level gems has to be done by the gem developers.

3.3. Main library gems

Main library gems are gems that depend too much on the framework. So they must be integrated into the framework. A basic graphic library like OpenGL or OpenSG could be such a main library gem. Another example is the ARToolKit. It's one possible technology that can be used for tracking. Also an object oriented abstraction mechanism of the tracking system is a main library gem. It is required for the exchange of the tracking technology, when we want to use a camera or magnetic based tracking system instead of an object recognition based tracking system. They are essential for the integrative design of the MR framework. For other kinds of main library gems we have to provide plug-in mechanisms as main library gems.
When we have to integrate a main library gem into the MR framework we must classified it into a group of the same functionality. For each gem group there must exist an abstraction mechanism to provide the exchangeability of most of the main library gems. The maintenance work for main library gem has to be done by the MR framework developers.
The main library gem groups and categories that can be identified yet and some gems that could be assigned to those classifications will be described in-depth in chapter 11 of this deliverable.

3.4. Inter-gem communication

To develop an application it is necessary to connect gems and create a gem network. So a communication between gems is required. This can be done by two ways. They are described in the next sections.

3.4.1. Direct communication

The first way is to call the methods of the gems direct without any overhead. These connections are fast and easy to understand.

3.4.2. Listener communication

The second communication mechanism is the listener concept known from the Java framework. It enables the monitoring of state changes of another object. Therefore a state machine gem and a state listener gem (as seen in Figure 2) must be provided by the framework. The state listener will be registered at the state machine for a specific trigger state. The listener is notified when the current state of the state machine is set to the trigger state.

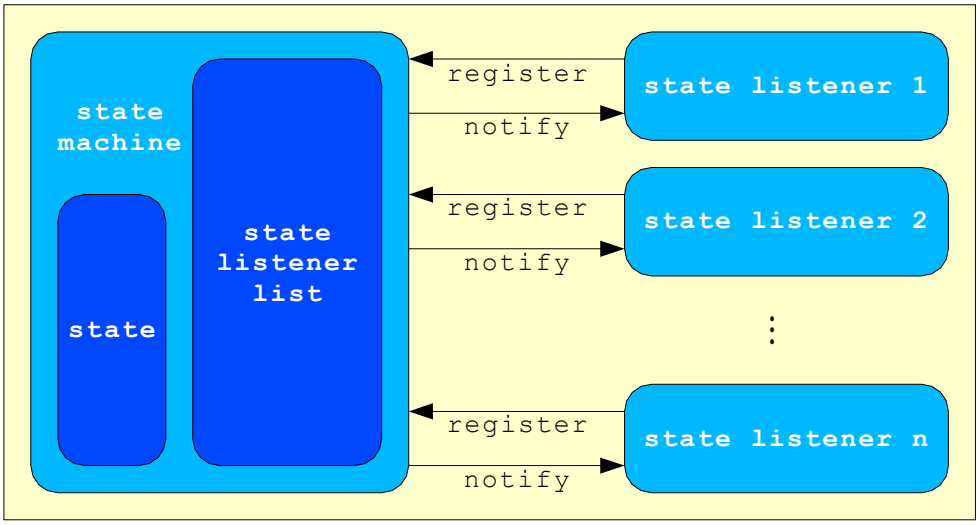

Figure 2: Listener concept

4. Component classification

In the next section we will classify components into three fields.

4.1. Wrapper components

The simplest components are the wrapper components. A single gem providing all the functionality of the component is wrapped by the component. This allows an application developer to use the functionality of the component as a gem. Gems are faster than components due to the overhead of their generic design.

4.2. Customized components

Generally customized components are implemented as a gem network. But they may also include existing components to reuse existing functionality that is not available as a gem but as a component. Customized components reflect the functionality required by the application author.

4.3. Composed components

It is often possible to provide new functionality by connecting components into a component network. A component developer would create a new component by using such a component network. A composed component allows the application author to create new components by his own. The composed component wraps the component network and provides the new functionality as a single component.

5. Properties

5.1. Typing

A generic communication between components must have a generic and well-defined data format. The beans concept [Hamilt02] uses the so-called properties for configuration of objects and communication between objects. The properties, presented in the beans concept, have the possibility to use all the meta and reflection interfaces of Java.
In C++ we don't have the possibility to get all the meta information about classes and methods. So we must design a property concept that works without most of the meta information. On the one hand properties in Java can be of any java class, which describes structured data. This implies for us that we need a structured property type. On the other hand a property in java can be simple base data types like integer and float. So we also need a set of simple base data types. In Java you can use the vector class or an array to describe an array of properties. So we need a vector property type of a specific property type. Further we need a property type describing an object reference. Altogether we need four groups of property types:

- Simple base property types
- Structured base property type
- Vector type
- Reference property type

Some of these property types could be exported or imported. As seen in the overview of all property types in Figure 3 these property types are persistent property types. In contrast to these persistent properties a reference property cannot be exported or imported due to the nature of its content. In the following sections we describe the four fundamental groups of property types in detail.

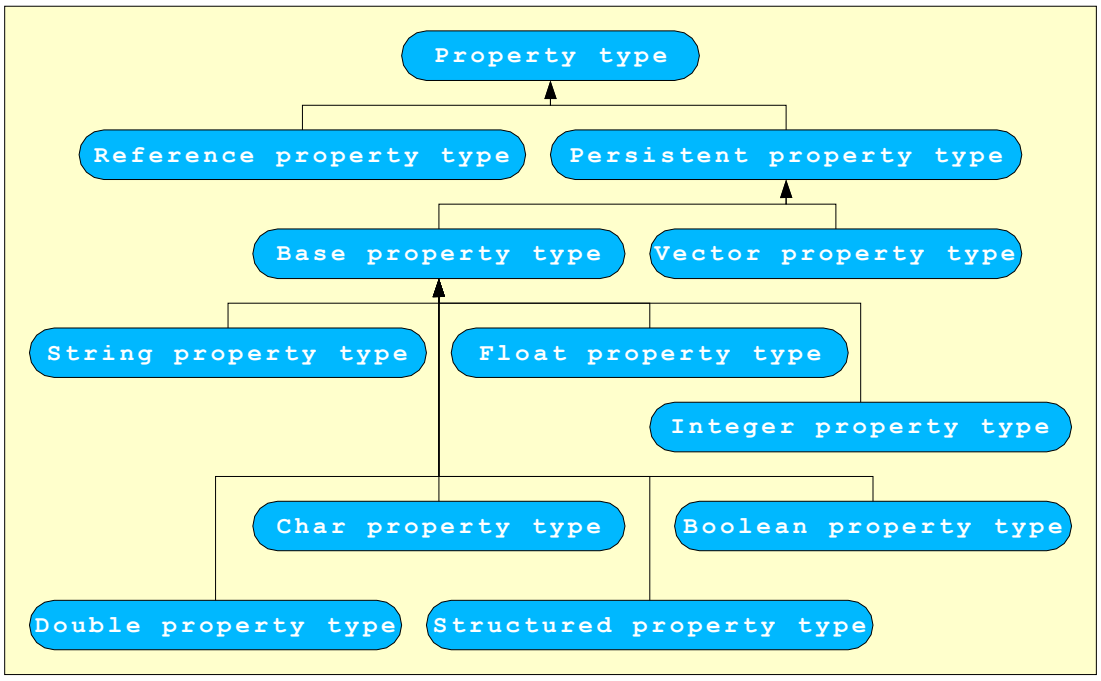

Figure 3: Overview of the property types

5.1.1. Reference property type

A reference property contains a reference pointer to a C++ object. The garbage collection of the reference pointer is not managed by the property concept. It is managed by the application. The reference property allows a fast exchange of information by using all the capabilities of C++ with a minimal memory overhead.

5.1.2. Simple base property types

The simple base property contains basic unstructured data and consists of following property types:

- Boolean property type
- Integer property type
- Float property type
- Double property type
- Character property type
- String property type

5.1.3. Structured base property type

A structured property type describes an associative array of property types. This means a structured property consists of property fields with specified persistent property types. Each field also could be a structured property. For example a structured property type for a 3D point will consist of x-, y- and z-coordinates that are float or double properties. Further a sphere structured property type contains a center (a 3D point structured property) and a radius (a float or double property).

5.1.4. Vector property type

A vector property type describes an array of a specified persistent property type. The length of a vector can grow. This means you can add persistent properties of the specified type. The vector property guarantees that all properties contained by the vector property are of the same persistent property type.

5.2. Property type manager

The property type manager (as seen in Figure 4) provides the necessary meta information required for the development of a generic authoring tool. It manages all available property types. All basic property types and the vector property type are singleton property types. They are automatically available at the property type manager. Structured property types must be registered at the property type manager. To get a structured property type (e.g. a 3D-point structured property type) you must lookup the structured property type at the property type manager by the name of the structured property type (e.g. 3D-point). The property type manager ensures the uniqueness of all property types. This uniqueness allows the developer to compare property types by comparing the reference pointers. So it will only cost a minimum of performance to guard the types of required and available properties. This will increase the stability and error permissiveness of the application. For example the type comparison will be used when two components are connected or a single component is configured.

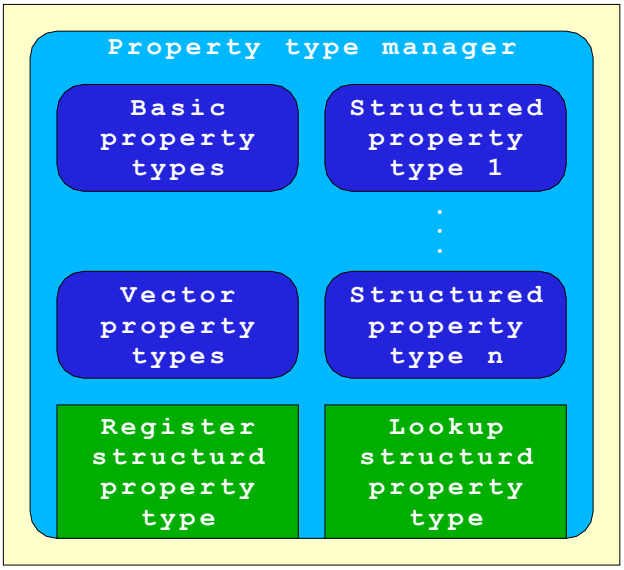

Figure 4: Property type manager

5.3. Persistence

To make a property persistent an export and import mechanism for properties is required. We recommend to ensure in an object oriented way that only properties will be exported or imported that can be made persistent. So a common base class persistent property will be required.

5.3.1. Export

Persistent properties are exported by a property writer (as seen in Figure 5). The property writer maps the property type of the persistent property to an *id*. The property type manager will do the mapping. The vector property type or simple basic property types like the integer property type must not be exported. Because they are integrated into the property concept and so they always must be mapped to the same id. In contrast to these integrated property type structured property types are registered at the property type manager. They have to be exported when it is necessary. To decide if it is necessary to export the structured property type the property writer must have a list of all exported structured property types. If the property type occurs in this list it is already exported and an id must be exported instead of the whole structured property type. Otherwise the property type and the mapped id must be exported. After the export of the persistent property type the property writer has to export the property by evaluating the information of the property type. This means:

- Simple basic properties are exported by exporting the simple data.
- Vector properties are exported by exporting the size, the specific property type of the vector property and recursively calling the export mechanism for all properties in the vector property.
- Structured properties are exported recursively calling the export mechanism for each field and exporting the name of the field.

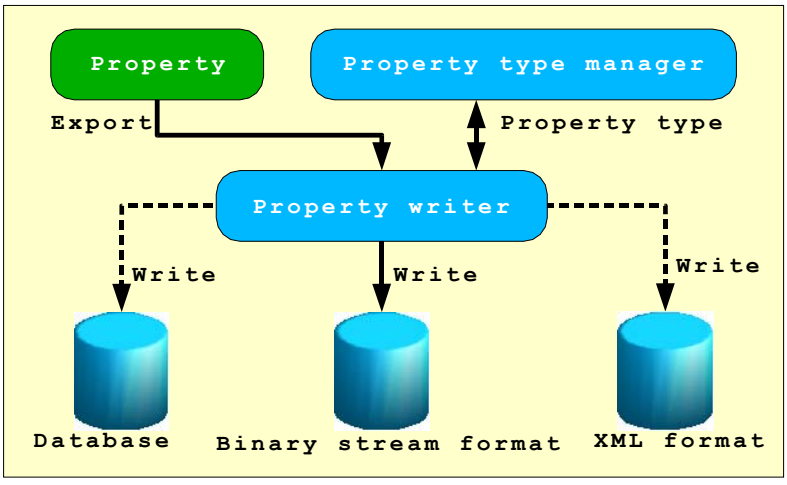

Figure 5: Exporting a property

The property writer is the abstraction of the different formats like:

- Database
- XML format (X3D for example)
- Binary stream format

So the property writer will be an abstract base class providing the interfaces required for the export of properties. To implement a property writer for a specific format you have to derive from this base class and implement the abstract interfaces.

5.3.2. Import

Persistent properties are imported by a property reader (as seen in Figure 6). The property reader imports the id of the property type of the persistent property and optionally the property type itself, if it occurs the first time. If the property type is imported it must be registered at the property type manager. Further the mapping of the id and the property type must be stored in the property reader, because the mapping between property type and id of the property type manager is not the same as the mapping at the export. If only the id is imported the stored mapping must be used to obtain the property type. After the property type is available a new property will be created and imported by evaluating the information of the property type. This means:

- Simple basic properties are imported by importing the simple data.
- Vector properties are imported by importing the size, the specific property type of the vector property and recursively calling the import mechanism for all properties given by the imported size.
- Structured properties are imported by recursively calling the import mechanism for each field and importing the name of the field.

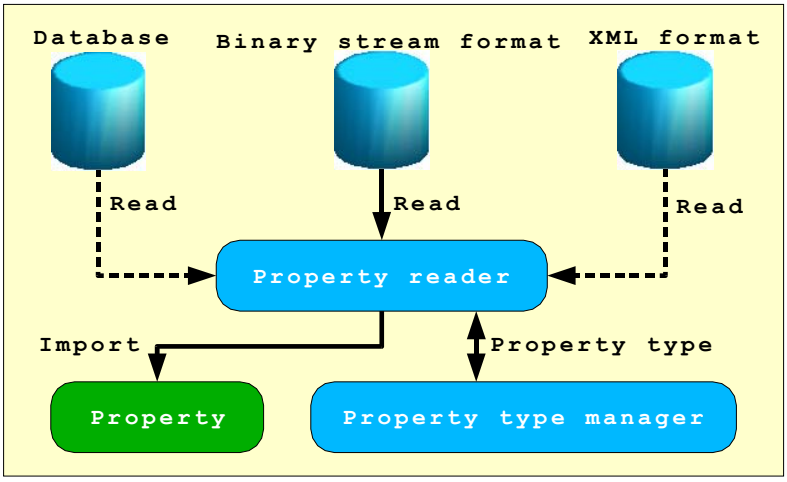

Figure 6: Importing a property

Like the property writer the property reader is also an abstract base class to provide an abstraction of the different formats. An implementation must be provided for each supported format.

5.4. Specification

In this section we specify the C++ implementation architecture of the properties. Therefore we begin with the inheritance hierarchy of the property values. The base class of each property value class is the class *Property* (as seen in). Two sub classes are derived from this base class:

- the class *RefernceProperty* is the implementation class of a property value that contains a C++ pointer with undefined type, which is in C++ realized with a void pointer. The semantic of the void pointer can only be evaluated by the knowledge of the application developer. This kind of property is important to wrap data that cannot be duplicated or to increase the performance, but it cannot be made persistent.
- the class *PeristentPropery* is an abstract base class for all properties values that can be made persistent. With this class it is possible to ensure at the compile time that only properties are used in interfaces, that can be exported. Without this base class it would be much harder to ensure the persistence of properties and failures would only be detected at the runtime.

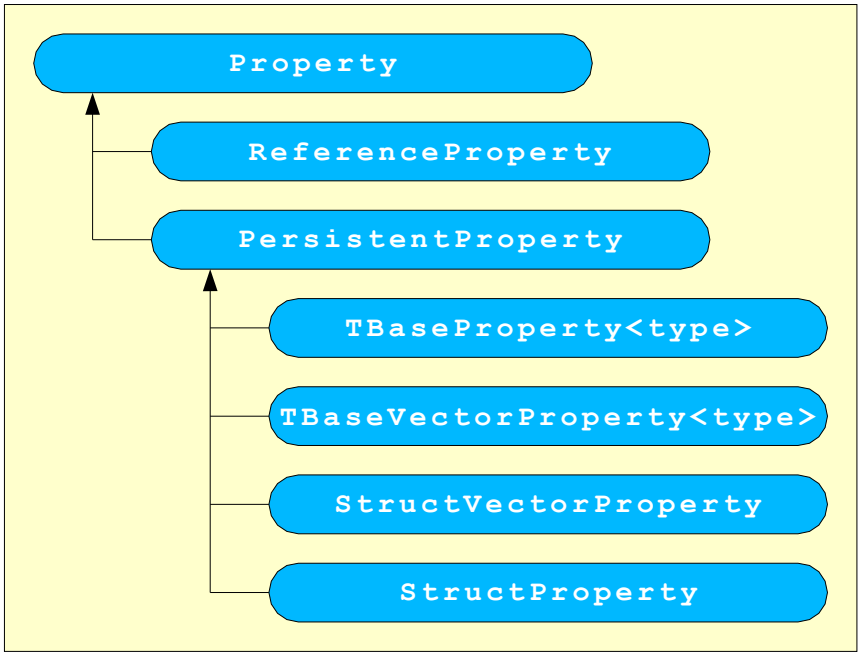

Figure 7: Property value class hierarchy

Four implementation classes are derived from the class PeristentPropery:

- the class *TBaseProperty<type>* is the template implementation class for all property values with unstructured base types like *bool*, *char*, int, *float*, *double* and *string*.
- the class *TBaseVectorProperty<type>* is the template implementation class for all vectors (an extensible array) of property values with unstructured base types like *bool*, *char*, int, *float*, *double* and *string*.

- the class StructProperty is the implementation class of all structured property values. It contains a set of named properties values which are at least of the class type *PeristentPropery*.
- the class StructVectorProperty is the implementation class of all vectors of structure property values. Such a vector of structured property values contains the structured property values and a base type for all property values. This means each property value must be at least of this property type. (IMPORTANT: The property type is not the same as the class type. The class types are managed by the compiler and the runtime environment and the property types are managed by the property type manager.)

Above we have introduced two template implementation classes. We have to explain what templates are, to understand what this exactly means. Templates are a very important part of the C++ programming language. Templates provide direct support for generic programming [STROUSTRUP]. This means you will use C++ language types and values as parameters to reduce the implementation work. A mathematic vector class for example with addition and subtraction methods can be implemented as a template.

Example:

```
template <class type> class TVector {
public:
  type x;
  type y;
  type z;

  inline TVector()
      : x(type(0.0)), y(type(0.0)), z(type(0.0)) {
  }

  inline TVector<type> &add(const TVector<type> &v) {
    x += v.x;
    y += v.y;
    z += v.z;
    return *this;
  }

  inline TVector<type> &sub(const TVector<type> &v) {
    x -= v.x;
    y -= v.y;
    z -= v.z;
    return *this;
  }
};
```

With this single template implementation class we have described the generic behavior of a simple mathematic vector class. It represents a set of classes with the same basic behavior. The detailed behavior depends on the parameter type. A integer and a float for example have different precisions. Further no executable code is created for the template. When you apply a parameter type to the template the compiler is able to create executable code and a implementation class for each kind.

Example:

```
TVector<int> integerVector;
TVector<float> floatVector;
TVector<double> doubleVector;
```

In the above example we instantiate a int, float and double vector object. Therefore the compiler creates three different classes and executable codes. So we have reduced the source code and therefore the possibility of undetected semantically errors but we have not reduced the size of the produced executable code. This sometimes leads to a underestimation of the size of the produce executable code.

In our case we have used templates to implement the classes for basic property values and vectors of basic property values as seen in Figure 8 and Figure 9.

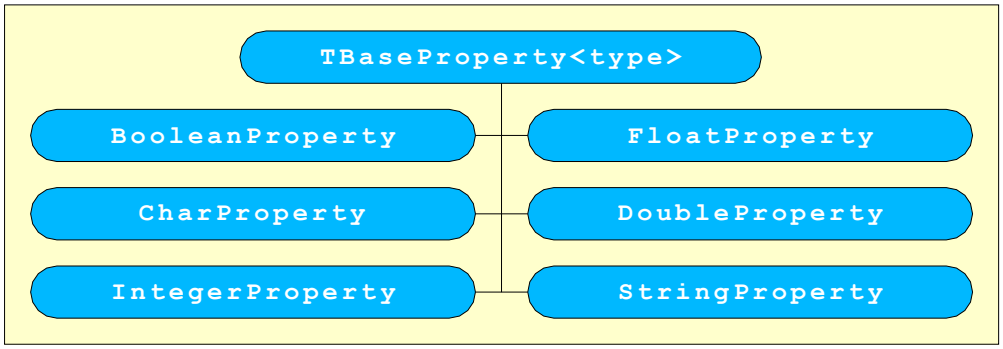

Figure 8: Basic property classes

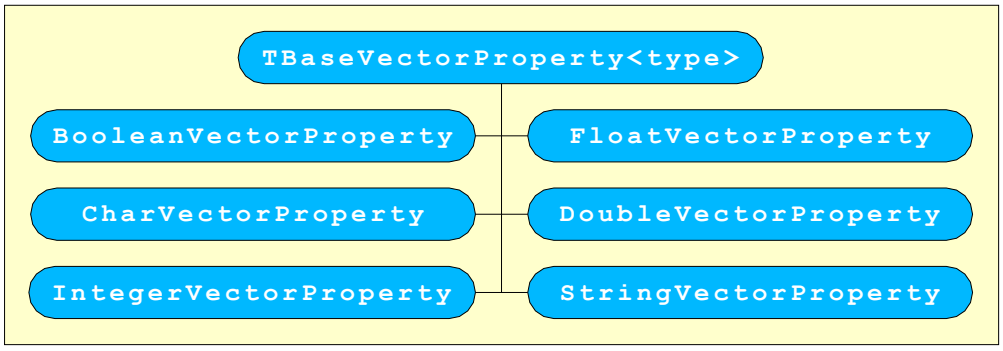

Figure 9: Basic vector property classes

Each property has a property type and therefore we need also a property type hierarchy as seen in Figure 10. The base class of each property type is the class *PropertyType*. Each base property type could be described with it. To describe all other property types we derive two class:

- the class *StructPropertType* describes the structure of a structured property. It contains the field names and property types of all property fields contained in the structured property. To inherit fields from another *StructPropertyType* a base property type also is contained.
- the class VectorPropertyType describes all types of vector properties. It contains the base property type of all elements. This means each element is of this type or of a derived one.

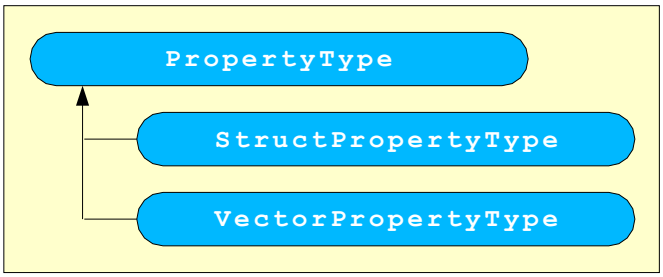

Figure 10: Property type class hierarchy

The property types are managed by the property type manager to reduce the set of existing property type instances to the minimal requires set. In Figure 11 we can see the relations between the class *PropertyTypeManager*, the class *Property*, the class *PropertyType* and its subclasses.

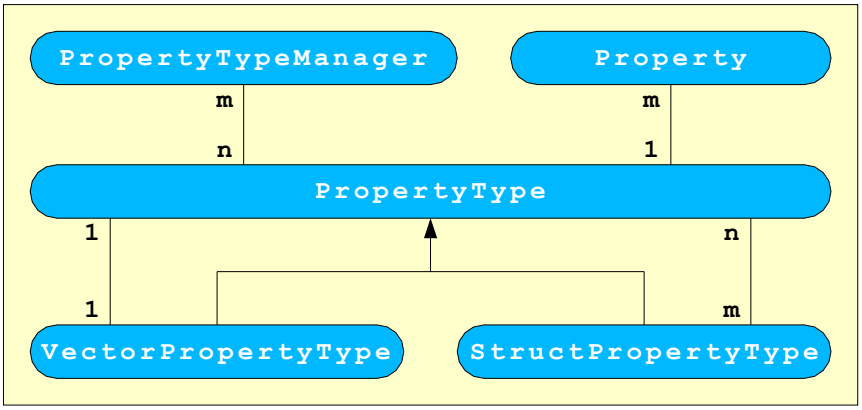

Figure 11: Relations between the class PropertyTypeManager and the class PropertyTypes

- Each property has exactly one property type to describe the semantic of its content and each property type can be the type of *m* or zero properties.
- For each property type (except the vector property types) there exists one vector property type containing this one property type.
- Each structured property type contains fields of *n* property types and each property type can be a field type in *m* structured property types.

- Each property type manager manages *n* (the minimal number of required property types) property types and each property type can be managed by *m* property type managers.

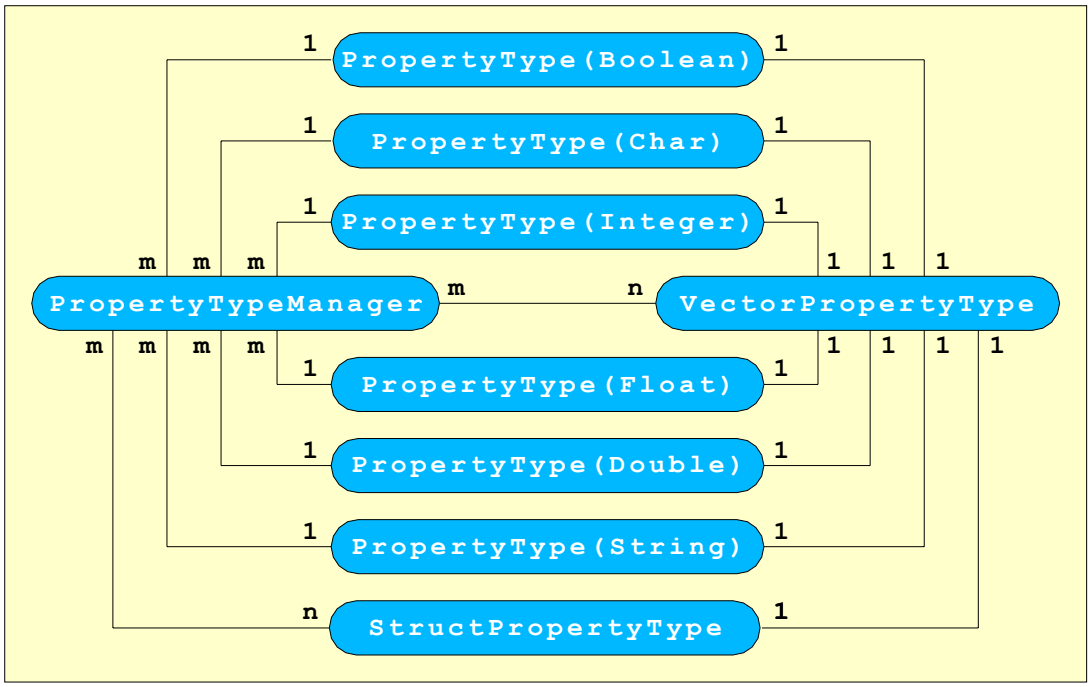

Figure 12: Detailed relations between property types and property type manager

The class *PropertyTypeManager* manages all property types of an application. The set of property types can be extended by registering structured property types at the property type manager. The encapsulation of the property type set in a property type manager instance allows us to handle several applications with different structured property types or structured property types with partially different structure, which may result from an modification of an application. Further the property type manager allows us the comparison of property type by comparing the reference pointers. This is possible due to two reasons:

- The first one is the registration and lookup of structured property types. As already mentioned the only way to extend the set of property types is to register structured property types at the property type manager. Further a property type manager provides a lookup interface for structured property types. This ensures that only one instance exists for the application that describes the property structure. This mechanism also handles the vector property types of structured property types. When a structured property type is registered the property type manager automatically adds a vector property type for this structured property type.
- The second one is the runtime wide uniqueness of basic property types and vector property types for basic property types. Therefore we have to examine details of the *n* to *m* relation between the class *PropertyTypeManager*, the class *PropertyType* and the class *VectorPropertyType* as seen in Figure 12. For each supported basic type a property type

instance is created (describing the base type as a kind identifier for *bool*, *char*, *int*, *float*, *double* and *string*). Further a vector property type is created for each base property type. The created instances are contained in the class *PropertyTypeManager* as static attributes of the class. Therefore each property type manager instance uses the same base property types.

In the next sections we specify the interfaces of all classes of the property architecture and their behavior. Further we give some examples how to create your own structured property types, how to use the property type manager and how to create properties of a specific type. The destructors will not be documented. They have to be virtual to enable the dynamic runtime type information which is necessary for the use of dynamic_cast instructions.

5.4.1. Property class

The class *Property* is the abstract base class of all Property classes. Properties contain data values with a specific semantic. Therefore each property has a property type. Further properties can be compared on equality. The interface is specified below:

```
class Property {
public:
  virtual ~Property();

  virtual const PropertyType *getType() const = 0;
  virtual bool equals(const Property *property) const = 0;
};
```

The '**`virtual const PropertyType *getType() const`**' method returns the property type of the property instance. It is an abstract virtual (dynamic binding in C++ will only used for virtual methods otherwise the compiler will create a static method call) method and must be implemented by each none abstract derived class. It doesn't modify the property instance and therefore it is *const*.

The '**`virtual bool equals(const Property *property) const`**' method return true when the as parameter *property* given property is equal to the *this* property instance and false otherwise. It also doesn't modify the property instance and therefore it is *const*.

5.4.2. ReferenceProperty class

Instances of the class *ReferenceProperty* wrap a reference with unspecified type (void pointer). The interface is specified below:

```
class ReferenceProperty : public Property {
public:
  ReferenceProperty(void *reference);

  virtual ~ReferenceProperty();

  virtual const PropertyType *getType() const;
  virtual bool equals(const Property *property) const;

  void *getReference() const;
  void setReference(void *reference);
};
```

The '**virtual const PropertyType *getType() const**' method returns the reference property type.

The '**virtual bool equals(const Property *property) const**' method return true when *property* is a instance of the class ReferenceProperty and the references are equal and false otherwise.

The '**void *getReference() const**' method returns the reference pointer wrapped by the reference property as void pointer. It doesn't modify the property instance and therefore it is *const*.

The '**void setReference(void *reference)**' method set the wrapped reference to the given reference value.

5.4.3. PersistentProperty class

The class PersistentProperty is the abstract base class of all persistence able properties.

```
class PersistentProperty : public Property {
public:
  virtual ~PersistentProperty();
};
```

5.4.4. TBaseProperty template class

The template implementation class *TBaseProperty<type>* is the implementation of all persistent base properties. Additionally to the derived interface it provides a get and set method for the property value, a constructor for creating a new base property with a given value and a copy constructor which creates a copy of a base property with the same class type.

```
template <class type> class TBaseProperty : public PersistentProperty {
public:
  TBaseProperty();
  TBaseProperty(const type &value);
  TBaseProperty(const TBaseProperty<type> &property);

  virtual ~TBaseProperty();

  virtual const PropertyType *getType() const;
  virtual bool equals(const Property *property) const;

  const type &getValue() const;
  void setValue(const type &value);
};
```

The '**virtual const PropertyType *getType() const**' method returns a base property type depending on which type is provided to the template.

The '**`virtual bool equals(const Property *property) const`**' method return true when *property* is a instance of the same property class and the values are equal. Otherwise false will be returned.

The '**`const type &getValue() const`**' method returns a *const* reference to the value contained in the base property. It also doesn't modify the property instance and therefore it is a *const* method. This allows the compiler better optimization of code containing method calls.

The '**`void setValue(const type &value)`**' method sets the value contained in the base property to the given value.

5.4.5. TBaseVectorProperty template class

The template implementation class *TBaseVectorProperty<type>* is the implementation of all persistent base vector properties. Additionally to the derived interface it provides a set of methods to manipulate the elements contained in the base vector property, a constructor for creating a new base vector property with a given list of values and a copy constructor which creates a copy of a base property with the same class type.

```
template <class type> class TBaseVectorProperty
    : public PersistentProperty {
public:
  TBaseVectorProperty();
  TBaseVectorProperty(int length, type *pvalues);
  TBaseVectorProperty(const TBaseVectorProperty<type> &property);

  virtual ~TBaseVectorProperty();

  virtual const PropertyType *getType() const;
  virtual bool equals(const Property *property) const;

  int size() const;
  const type &get(int index) const;
  void set(int index, const type &value);
  void insert(int index, type value);
  void remove(int index);
  void push(type value);
  type pop();
};
```

The '**`virtual const PropertyType *getType() const`**' method returns a base vector property type depending on which type is provided to the template.

The '**virtual bool equals(const Property *property) const**' method return true when *property* is a instance of the same property class and the comparison result of the two value arrays is equal. Otherwise false will be returned.

The '**int size() const**' method returns the actual number of the values contained in the vector.

The '**const type &get(int index) const**' method returns a *const* reference of the value contained in the vector at the position specified by the given index. The allowed index range is from 0 to one below the number of values in the vector (0 .. size – 1). An assert exception is thrown when the given index is out of range.

The '**void set(int index, const type &value)**' method sets the value in the vector at the specified index to the given value. The allowed index range is from 0 to one below the number of values in the vector (0 .. size – 1). An assert exception is thrown when the given index is out of range.

The '**void insert(int index, type value)**' method inserts the given value of the specified index into the value array of the vector. The allowed index range is from 0 to the number of values in the vector (0 .. size). This means you also can insert at the end of the vector. An assert exception is thrown when the given index is out of range.

The '**void remove(int index)**' method removes the value in the vector at the specified position. The allowed index range is from 0 to one below the number of values in the vector (0 .. size – 1). An assert exception is thrown when the given index is out of range.

The '**void push(type value)**' method adds the given value to the end of the vector like a push operation on a stack.

The '**type pop()**' method removes the last value from the vector and returns the removed value. An assert exception is thrown when the vector is empty.

5.4.6. StructProperty class

Instances of the class *StructProperty* represents a associative array of properties. The number of properties and their names and types are defined by a structured property type. It provides interfaces to access and manipulate the properties by their name. Special interfaces are provided to ensure a fast access to basic and vector properties.

```
class StructProperty : public PersistentProperty {
public:
  StructProperty(const string &structName,
                 const PropertyTypeManager *propertyTypeManager = NULL);

  virtual ~StructProperty();

  virtual const PropertyType *getType() const;
  virtual bool equals(const Property *property) const;

  bool contains(const string &name) const;
```

```
  void setProperty(const string &name, PersistentProperty *property);
  const PersistentProperty *getProperty(const string &name) const;

  void setBoolean(const string &name, bool value);
  void setChar(const string &name, char value);
  void setInteger(const string &name, int value);
  void setFloat(const string &name, float value);
  void setDouble(const string &name, double value);
  void setString(const string &name, const string &value);

  bool getBoolean(const string &name) const;
  char getChar(const string &name) const;
  int getInteger(const string &name) const;
  float getFloat(const string &name) const;
  double getDouble(const string &name) const;
  const string &getString(const string &name) const;
  StructProperty *getStruct(const string &name);
  BooleanVectorProperty *getBooleanVector(const string &name);
  CharVectorProperty *getCharVector(const string &name);
  IntegerVectorProperty *getIntegerVector(const string &name);
  FloatVectorProperty *getFloatVector(const string &name);
  DoubleVectorProperty *getDoubleVector(const string &name);
  StringVectorProperty *getStringVector(const string &name);
  StructVectorProperty *getStructVector(const string &name);
};
```

The '**virtual const PropertyType *getType() const**' method returns a structured property type containing the fieldnames and types.

The '**virtual bool equals(const Property *property) const**' method return true when *property* is a structured property with the same structured property type and equal content. Otherwise false will be returned.

The '**bool contains(const string &name) const**' method return true when the structured property type defines that the structured property must contain a property with the given name. Otherwise false will be returned.

The '**void setProperty(const string &name, PersistentProperty *property)**' method sets the property associated with the name to the given value. An assert exception is thrown when:

- the fieldname is not defined in the structured property type,
- the property type of the given property is not the same as the defined one or
- the given property is null.

The '**const PersistentProperty *getProperty(const string &name) const**' method returns the property associated with the given name. An assert exception is thrown when the fieldname is not defined in the structured property type.

The '`void setBoolean(const string &name, bool value)`' method sets the boolean property associated with the name to the given value. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the boolean property type.

The '`void setChar(const string &name, char value)`' method sets the character property associated with the name to the given value. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the character property type.

The '`void setInteger(const string &name, int value)`' method sets the integer property associated with the name to the given value. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the integer property type.

The '`void setFloat(const string &name, float value)`' method sets float the property associated with the name to the given value. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the float property type.

The '`void setDouble(const string &name, double value)`' method sets double the property associated with the name to the given value. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the double property type.

The '`void setString(const string &name, const string &value)`' method sets the string property associated with the name to the given value. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the string property type.

The '`bool getBoolean(const string &name) const`' method returns the boolean property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the boolean property type.

The '`char getChar(const string &name) const`' method returns the character property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the character property type.

The '`int getInteger(const string &name) const`' method returns the integer property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the integer property type.

The '`float getFloat(const string &name) const`' method returns the float property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the float property type.

The '`double getDouble(const string &name) const`' method returns the double property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the double property type.

The '`const string &getString(const string &name) const`' method returns the string property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the string property type.

The '`bool getBoolean(const string &name) const`' method returns the boolean property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the boolean property type.

The '`char getChar (const string &name) const`' method returns the character property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the character property type.

The '`int getInteger (const string &name) const`' method returns the integer property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the integer property type.

The '`float getFloat (const string &name) const`' method returns the float property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the float property type.

The '`double getDouble (const string &name) const`' method returns the double property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the double property type.

The '`const string &getString (const string &name) const`' method returns the string property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the string property type.

The '`StructProperty *getStruct(const string &name)`' method returns a reference pointer pointing to the structured property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not a structured property type.

The '`BooleanVectorProperty *getBooleanVector(const string &name)`' method returns a reference pointer pointing to the boolean vector property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the boolean vector property type.

The '`CharVectorProperty *getCharVector(const string &name)`' method returns a reference pointer pointing to the character vector property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the character vector property type.

The '`IntegerVectorProperty *getIntegerVector(const string &name)`' method returns a reference pointer pointing to the integer vector property associated with the given name. An assert

exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the integer vector property type.

The '**`FloatVectorProperty *getFloatVector(const string &name)`**' method returns a reference pointer pointing to the float vector property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the float vector property type.

The '**`DoubleVectorProperty *getDoubleVector(const string &name)`**' method returns a reference pointer pointing to the double vector property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the double vector property type.

The '**`StringVectorProperty *getStringVector(const string &name)`**' method returns a reference pointer pointing to the string vector property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not the string vector property type.

The '**`StructVectorProperty *getStructVector(const string &name)`**' method returns a reference pointer pointing to the structured vector property associated with the given name. An assert exceptions is thrown when the fieldname is not defined in the structured property type or the defined property type is not a structured vector property type.

5.4.7. StructVectorProperty class

Instances of the class *StructVectorProperty* contains several structured properties of a specified base type. Additionally to the derived interface it provides a set of methods to manipulate the elements contained in the structured vector property, a constructor for creating a empty structured vector property and a copy constructor which creates a copy of a base property with the same class type.

```
class StructVectorProperty : public PersistentProperty {
public:
  StructVectorProperty(const string &structName,
                       const PropertyTypeManager *propertyTypeManager =
                       NULL);
  StructVectorProperty(const StructVectorProperty &property);

  virtual ~StructVectorProperty();

  virtual const PropertyType *getType() const;
  virtual bool equals(const Property *property) const;

  int size() const;
  StructProperty *at(int index);
  const StructProperty *get(int index) const;
  void set(int index, StructProperty *value, bool deleteValue = true);
  void insert(int index, StructProperty * value);
  void remove(int index, bool deleteValue = true);
  void push(StructProperty * value);
  StructProperty *pop();
```

```
};
```

The '**virtual const PropertyType *getType() const**' method returns a structured vector property type containing the structured property type of the vector.

The '**virtual bool equals(const Property *property) const**' method return true when *property* is a structured vector property with the same structured property type and equal content. Otherwise false will be returned.

The '**int size() const**' method returns the number of structures properties that are currently contained in the vector.

The '**StructProperty *at(int index)**' method returns the structured property at the specified index, which should be used when you want to modify the structured property otherwise you should use the get method. An assert exception is thrown when the index is out of range (index >= size).

The '**const StructProperty *get(int index) const**' method returns the structured property at the specified index as const, which means that you cannot modify its content. An assert exception is thrown when the index is out of range (index >= size).

The '**void set(int index, StructProperty *value, bool deleteValue = true)**' method sets the structured property at the specified index to the given value. The structured property previously stored at the index is deleted, when the *deleteValue* parameter is set to true, which is the default value. An assert exception is thrown when:

- the index is out of range (index >= size),
- the given value is null or
- the structured property type of the given value is not derived from the base structured property type of the vector.

The '**void insert(int index, StructProperty * value)**' method inserts the given structured property value into the vector at the specified index. The insertion at the end of the vector (index = size) is allowed. An assert exception is thrown when:

- the index is out of range (index > size),
- the given value is null or
- the structured property type of the given value is not derived from the base structured property type of the vector.

The '**void remove(int index, bool deleteValue = true)**' method removes the structured property from the vector at the specified index. An assert exception is thrown when the index is out of range (index >= size).

The '**void push(StructProperty * value)**' method appends the given structured property value at the end of the vector like a push on a stack. An assert exception is thrown when:

- the given value is null or

- the structured property type of the given value is not derived from the base structured property type of the vector.

The '**`StructProperty *pop()`**' method removes the last structured property value from the vector. An assert exception is thrown when the vector is empty.

5.4.8. PropertyType class

An instance of the class *PropertyType* describes the type of a property. The description is done by the kind of the property type. It distinguishes several basic kinds:

- *REFERENCE_PROPERTY* describes the reference property type, which doesn't require any further type description.
- *BOOLEAN_PROPERTY* describes the boolean property type, which doesn't require any further type description.
- *CHAR_PROPERTY* describes the character property type, which doesn't require any further type description.
- *INTEGER_PROPERTY* describes the integer property type, which doesn't require any further type description.
- *FLOAT_PROPERTY* describes the float float type, which doesn't require any further type description.
- *DOUBLE_PROPERTY* describes the double property type, which doesn't require any further type description.
- *STRING_PROPERTY* describes the string property type, which doesn't require any further type description.
- *STRUCT_PROPERTY* describes the reference property type, which requires further type description containing the field names and types.
- *VECTOR_PROPERTY* describes the reference property type, which requires further type description containing the property type of the vector property type.

```
class PropertyType {
public:
  typedef enum {
    REFERENCE_PROPERTY = 0,
    BOOLEAN_PROPERTY,
    INTEGER_PROPERTY,
    FLOAT_PROPERTY,
    DOUBLE_PROPERTY,
    CHAR_PROPERTY,
    STRING_PROPERTY,
    STRUCT_PROPERTY,
    VECTOR_PROPERTY
  } Kind;

public:
  virtual ~PropertyType();

  virtual bool isTypeOf(const PropertyType *type) const;

  PropertyType::Kind getKind() const;
```

```
  PropertyTypeManager *getPropertyTypeManager();
};
```

The '**virtual bool isTypeOf(const PropertyType *type) const**' method returns true when the given property type is the same property type as this one or a base property type of this one. Otherwise false is returned.

The '**PropertyType::Kind getKind() const**' method returns the kind of property type.

5.4.9. VectorPropertyType class

The class *VectorPropertyType* describes the all vector property types containing the element property type.

```
class VectorPropertyType : public PropertyType {
public:
  VectorPropertyType(const PropertyType *propertyType,
                     PropertyTypeManager *propertyTypeManager = NULL);

  virtual ~VectorPropertyType();

  virtual bool isTypeOf(const PropertyType *type) const;

  const PropertyType *getPropertyType() const;
};
```

The '**virtual bool isTypeOf(const PropertyType *type) const**' method returns true when the given property type is the same vector property type as this one and the element property types are also the same. Otherwise false is returned.

The '**const PropertyType *getPropertyType() const**' method returns the element property type.

5.4.10. StructPropertyType class

An instance of the class *StructPropertyType* describes a structured property type by specifying all field names and types of the structured property type and its base structured property type. It provides interfaces to manipulate the structure and to access all field information.

```
class StructPropertyType : public PropertyType {
public:
  StructPropertyType(const string &structName,
                     PropertyTypeManager *propertyTypeManager = NULL);
  StructPropertyType(const string &structName,
                     const string &baseStructName,
                     PropertyTypeManager *propertyTypeManager = NULL);

  virtual ~StructPropertyType();

  virtual bool isTypeOf(const PropertyType *type) const;
```

```
  const StructPropertyType *getBaseType() const;
  const string &getStructName() const;
  bool isFieldDefined(const string &fieldName) const;
  const vector<string> &getFieldNames() const;
  const PropertyType *getFieldType(const string &fieldName) const;
  void addField(const string &fieldName, const PropertyType *type);
  void changeField(const string &fieldName, const PropertyType *type);
  void removeField(const string &fieldName);
};
```

The '**virtual bool isTypeOf(const PropertyType *type) const**' method returns true when the given property type is the same structured property type as this one or a base structured property type of this one. Otherwise false is returned.

The '**const StructPropertyType *getBaseType() const**' method returns the base structured property type or null if no base structured property type is define.

The '**const string &getStructName() const**' method returns the name of the structured property type.

The '**const bool isFieldDefined(const string &fieldName) const**' method returns true when the field with the specified name is defined in the structured property type. Otherwise false is returned.

The '**const vector<string> &getFieldNames() const**' method returns the field names that are defined in the structured property type.

The '**const PropertyType *getFieldType(const string &fieldName) const**' method returns the property type of the field specified by the field name. An assert exception is thrown when the field is not defined in the structured property type.

The '**void addField(const string &fieldName, const PropertyType *type)**' method adds a new field with the specified field name and type. An assert exception is thrown when:

- the given type is null or
- the field is already defined in the structured property type.

The '**void changeField(const string &fieldName, const PropertyType *type)**' method changes the property type of the field specified by the field name to the given property type. An assert exception is thrown when:

- the given type is null or
- the field is not defined in the structured property type.

The '**void removeField(const string &fieldName)**' method removes the field definition specified by the field name from the structured property type. An assert exception is thrown when the field is not defined in the structured property type.

5.4.11. PropertyTypeManager class

An instance of the class *PropertyTypeManager* manages all structured property types of an application. Two applications exist concurrently must use two property type managers to ensure the separation of the applications. An application A for example defines the camera property by an orientation matrix and a position. Another application B could redefine the camera property by a position and a roll, pitch and yaw angle for the orientation. This two applications would have two different definitions for the camera property. This is the reason why each application need its own property type manager. Additionally there always exists a default property type manager which should be used, when only one application is active.

```
class PropertyTypeManager {
public:
  static PropertyTypeManager *getDefaultPropertyTypeManager();

  static const PropertyType *getBooleanPropertyType();
  static const PropertyType *getCharPropertyType();
  static const PropertyType *getIntegerPropertyType();
  static const PropertyType *getFloatPropertyType();
  static const PropertyType *getDoublePropertyType();
  static const PropertyType *getStringPropertyType();
  static const PropertyType *getBooleanVectorPropertyType();
  static const PropertyType *getCharVectorPropertyType();
  static const PropertyType *getIntegerVectorPropertyType();
  static const PropertyType *getFloatVectorPropertyType();
  static const PropertyType *getDoubleVectorPropertyType();
  static const PropertyType *getStringVectorPropertyType();

  PropertyTypeManager();

  virtual ~PropertyTypeManager();

  bool contains(const string &structName) const;
  const vector<string> &getStructPropertyTypeNames() const;
  const StructPropertyType *getStructType(const string &name) const;
  const VectorPropertyType *getStructVectorType(const string &name) const;
  void registerStructType(StructPropertyType *structType);
};
```

The '**`static PropertyTypeManager *getDefaultPropertyTypeManager()`**' method returns the default property type manager, which should be used for standalone applications. It is static, which means it can be invoked without an object by calling the method with the class name as qualifier like *`PropertyTypeManager::getDefaultPropertyTypeManager()`* for example.

The '**`static const PropertyType *getBooleanPropertyType()`**' method returns the boolean property type which is unique for the whole runtime environment.

The '**`static const PropertyType *getCharPropertyType()`**' method returns the character property type which is unique for the whole runtime environment.

The '**`static const PropertyType *getIntegerPropertyType()`**' method returns the integer property type which is unique for the whole runtime environment.

The '`static const PropertyType *getFloatPropertyType()`' method returns the float property type which is unique for the whole runtime environment.

The '`static const PropertyType *getDoublePropertyType()`' method returns the double property type which is unique for the whole runtime environment.

The '`static const PropertyType *getStringPropertyType()`' method returns the string property type which is unique for the whole runtime environment.

The '`static const PropertyType *getBooleanVectorPropertyType()`' method returns the boolean vector property type which is unique for the whole runtime environment.

The '`static const PropertyType *getCharVectorPropertyType()`' method returns the character vector property type which is unique for the whole runtime environment.

The '`static const PropertyType *getIntegerVectorPropertyType()`' method returns the integer vector property type which is unique for the whole runtime environment.

The '`static const PropertyType *getFloatVectorPropertyType()`' method returns the float vector property type which is unique for the whole runtime environment.

The '`static const PropertyType *getDoubleVectorPropertyType()`' method returns the double vector property type which is unique for the whole runtime environment.

The '`static const PropertyType *getStringVectorPropertyType()`' method returns the string vector property type which is unique for the whole runtime environment.

The '`bool contains(const string &structName) const`' method returns true when a structured property type with the given name is defined in this property type manager. Otherwise false is returned.

The '`const vector<string> &getStructPropertyTypeNames() const`' method returns the name list of structured property types that are defined in this property type manager.

The '`const StructPropertyType *getStructType(const string &name) const`' method returns the structured property type that is specified by the given name. An assert exception is thrown when no structured property type with the given name is defined in this property type manager.

The '`const VectorPropertyType *getStructVectorType(const string &name) const`' method returns the structured vector property type that is specified by the given name. An assert exception is thrown when no structured vector property type with the given name is defined in this property type manager.

The '`void registerStructType(StructPropertyType *structType)`' method registers the definition of the given structure property type in this property type manager. Further the corresponding structured vector property type for the given structured property type is created and registered in this

property type manager. An assert exception is thrown when the structured property type is null or a structured property type with the same name is already defined in this property type manager.

5.4.12. PropertyReader class

The class *PropertyReader* is the abstract base interface of all property reader implementations.

```
class PropertyReader {
public:
  virtual ~PropertyReader();

  virtual PersistentProperty *read() = 0;
};
```

The '**`virtual PersistentProperty *read() = 0`**' method is abstract and has to be implemented by each property reader. It reads and returns a persistent property. An assert exception is thrown in any failure case.

5.4.13. PropertyWriter class

The class *PropertyWriter* is the abstract base interface of all property writer implementations.

```
class PropertyWriter {
public:
  virtual ~PropertyWriter() = 0;

  virtual void write(const PersistentProperty *property) = 0;
};
```

The '**`virtual void write(const PersistentProperty *property) = 0`**' method is abstract and has to be implemented by each property writer. It writes the given persistent property. An assert exception is thrown in any failure case.

5.4.14. Examples

In this section we will show how to use properties. First we begin with the creation of a simple basic type properties like a boolean property:

```
BooleanProperty b;  // create the property
b.setValue(true);   // set the value
if (b.getValue()) { // get the value
  // do something
}
```

The next example will show the use of vector properties like a double vector property:

```
DoubleVectorProperty v; // create the property
v.push(2.0);            // insert a value at the end of the vector
v.insert(0, 1.0);       // insert a value at the begin of the vector
for (int i = 0; I < v.size(); ++I) {
  v.get(i);             // get the value at the index and do something
}
```

This examples have shown the use of simple base property types. In the next example we will show the creation of a new structured property type.

```
StructPropertyType t1("MyType1");
t.addField("myField1", PropertyTypeManager::getBooleanPropertyType());

// ... registration of t1

StructPropertyType t2("MyType2", "MyType1");
t.addField("myField2", PropertyTypeManager::getDoubleVectorPropertyType());
```

We have created a new structured property type with the name “MyType” containing the definition of a field with the name “myField1”, which is defined as a boolean property. Then we have extended the “MyType1” into “MyType2”. It extends the base type by the new field “myField2”, which is defined as a double vector property. In the next example we will show how you can register the property type at a property type manager and create new instances of a structured property type.

```
PropertyTypeManager *m =
    PropertyTypeManager::getDefaultPropertyTypeManager()

StructPropertyType t1("MyType1");
t.addField("myField1", PropertyTypeManager::getBooleanPropertyType());

m.registerStructType(t1);

StructPropertyType t2("MyType2", "MyType1");
t.addField("myField2", PropertyTypeManager::getDoubleVectorPropertyType());

m.registerStructType(t2);

StructProperty p1("MyType1");
p1.setBoolean("myField1", true);

StructProperty p2("MyType2");
p1.setBoolean("myField1", false);
p1.getDoubleVector("myField2").push(1.0);
```

6. Components

6.1. Inter-object communication concepts

Inter-object communication is often done by directly calling methods of objects. This is indeed a very fast and easy to understand way of communication between two objects. In the C++ standard there is only a minimum of meta information and interfaces available [Strous00], which does not include a concept like the Java reflection API (application programmer interface) [Green02]. For the development of authoring frameworks or applications a subset of such a method call abstraction by meta interfaces is mandatory. Due to this lack of such a reflection API in C++ it is unrealistic to develop such authoring frameworks or applications in C++ by using direct method calls for inter-object communication.

6.1.1. Signals and slots

In the GUI (graphical user interface) library QT Trolltech Inc. uses a design pattern called signals and slots for inter-object communication [Trollt02]. This is a concept for connecting objects, which is easy to understand and to use. Trolltech Inc. also presents a generic authoring tool based on the signals and slots. The implementation of this concept is achieved by a MOC (meta object compiler). This MOC provides all the necessary meta information needed by the signals and slots. But the implementation of this concept only allows the authoring tool to create source code for the application. For modifications on the application it is inevitable to recompile the sources.

6.1.2. Beans and properties

Another concept for authoring tools is the beans concept [Hamilt02]. It is a software component model for Java, which allows the generic manipulation of such components. This is done by defining a set of properties for each bean. The Java reflection API provides the interfaces to manipulate these properties and to connect beans by using properties. Sun demonstrates the applicability of the beans concept for authoring tools by a small application called BeanBox [SunMic02a].

6.1.3. State oriented listeners

For the communication between objects we also need a mechanism allowing the objects to react on a specific situation like a pressed button or a collision with a chair. Java affords the listener concept for this requirement [SunMic02b]. This mechanism is a simplified version of the well-known MVC (model-view-controller) design pattern [GamHel95]. It provides a mechanism to notify views about the change of a specific model. This model change can also be considered as a change of the object state. This interpretation is realized in the state machine of Rabin [Rabin00]. Rabin's state machine uses a centralized mechanism routing state change messages between objects. Such a centralized architecture is a restriction for the future design. Moreover, it becomes a bottleneck in the future. With a decentralized architecture the prevention of bottlenecks would be possible. A central object used for delegation and rooting can simulate the centralized architecture.

6.2. Inter-Component communication

We have decided to develop a new architecture, which combines the advantages of the above concepts into a fast, easy to use and generic concept of inter-object communication for C++. First of all we will look at the requirements for a component. Components have to be:

- Connectable by a simple and easy to use mechanism
- Configurable in a generic way
- State oriented
- Instancable by a classifying name
- Persistent

6.2.1. In- and out-slots

Compared with the design of QT objects, which uses signals and slots, components (as seen in Figure 13) have two slot types (in- and out-slots). In fact this is a more symmetric naming than signals and slots. In contrast to the QT design, component slots are named dynamically and not statically by the compiler. So it becomes possible to access the in- and out-slots by their names and to show a list of available slots for each component.

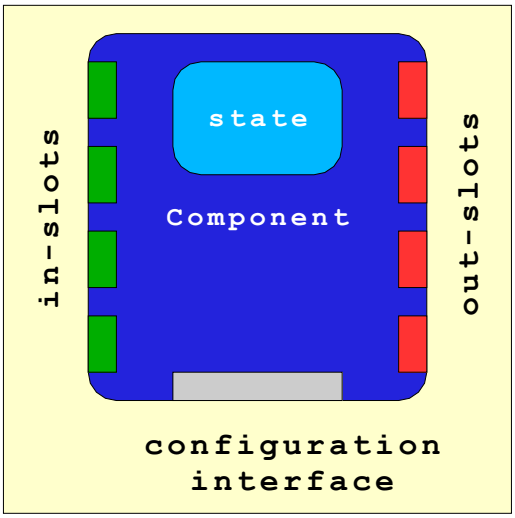

Figure 13: Component interfaces

6.2.2. Typing and connectability

We recommend the use of properties for in- and out-slots. This allows us the typing of each slot. Only slots of the same type are connectable. This increases the usability, flexibility and extensibility of a generic authoring tool. The comparison of the types is only a basic behavior of the components concept to decide if in- and out-slot are connectable. An additional interface provides the component developer with a way to deny the connection of an in- and out-slot of the same type. This allows it to build high-level connectivity rules which base on the component specific semantic of properties.

6.2.3. Connections

As already mentioned all slots have a name. So basically a connection between two components consists of the in- and out-slot names and the references of the emitting and receiving component (as seen in the component network in Figure 14). A polling mechanism will cost unnecessarily performance and implies a polling thread, which will lead to a complex synchronization strategy. Therefore, a connection also needs a trigger state for the emitting component. This implies a well-formed state interface, which has to be part of the component interfaces. It is recommended to map the slot name to a slot id. This will increase the performance due to the high costs of comparing strings and the fast comparison of numbers. The transfer of the data will be realized by telling the emitter component to emit the out-slot data property and providing it to the receiver components in-slot.

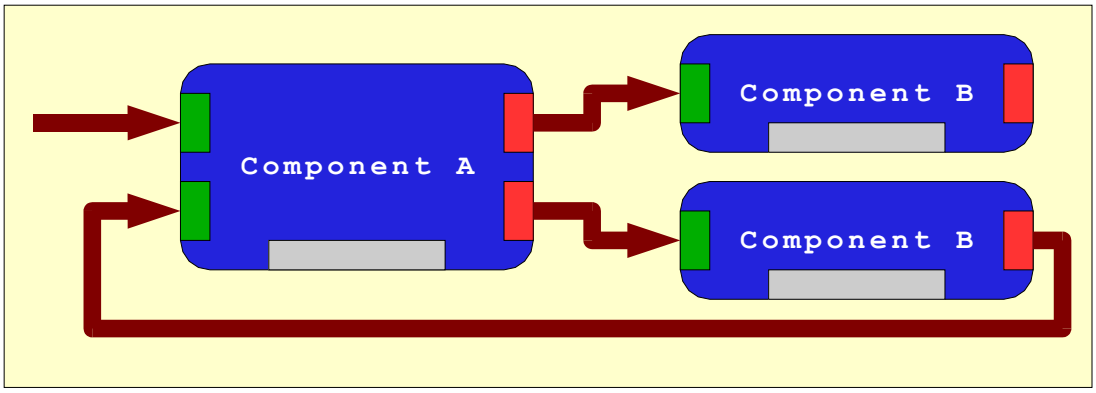

Figure 14: Component connections

6.3. Configuration

We recommend the configuration of components with properties. Using the information of the property type will avoid programming failures or enables at least the location of them as soon and fast as possible. Further the use of a generic data structure like the properties enables the creation of a generic configuration component.

6.4. Composed components

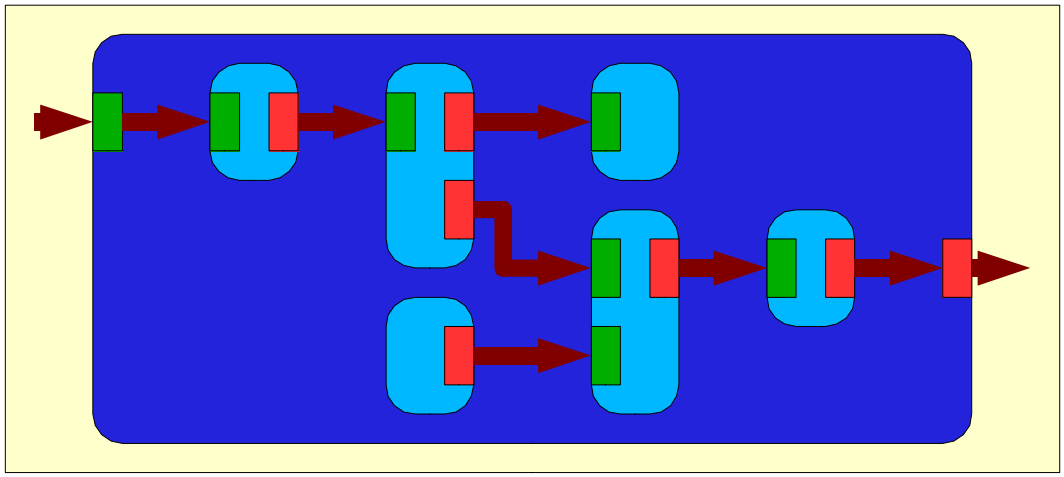

Figure 15: Composed component

A component can be considered as a black box. It has in- and out-slot, a current state visible to other components and it has to be configurable. A component network often provides a new functionality. When this functionality is required very often a component should be developed, which provides this functionality. This will increase the reuse of functionality and reduce the complexity of the application. So why should an application author ask the component developer to develop a component, which makes the same as his component network? It should be possible to select such a component network put it into a composed component, which abstracts the component network. The application author only has to select in-, out-slot and configuration interfaces, which have to be exported. The possible visible states are defined by the exported out-slots. To provide a better usability of this composed component the slot names of the composed component should be changed. Further a name must be defined for the component.

6.5. Component manager

In Java there is an easy mechanism to create an instance of a specific class. You can lookup a class by its name and call a constructor to create a new instance of this class. In C++ this is not possible. So we need an object, which manages the creation and the lifetime of the components. We call the object a component manager (as seen in Figure 16). You can register a component prototype at the component manager by its name. This also includes a composed component that's why a name must be assigned to a composed component. To create a new instance of a component, each component must have an interface to create a clone of it. The original prototype must not be modified after the registration to ensure the same default behavior for each instance. The component manager is also the central storage of all components used in the application. This will bypass the missing garbage collection in C++. Deleting a component will also delete the connections it is assigned to.
To make fast state comparison possible the component manager is also storage of all system or application wide states. Each state used in application components must have a name and must be registered at the component manager. The component manager must not be a singleton to ensure that more than one application can be loaded without conflicts.

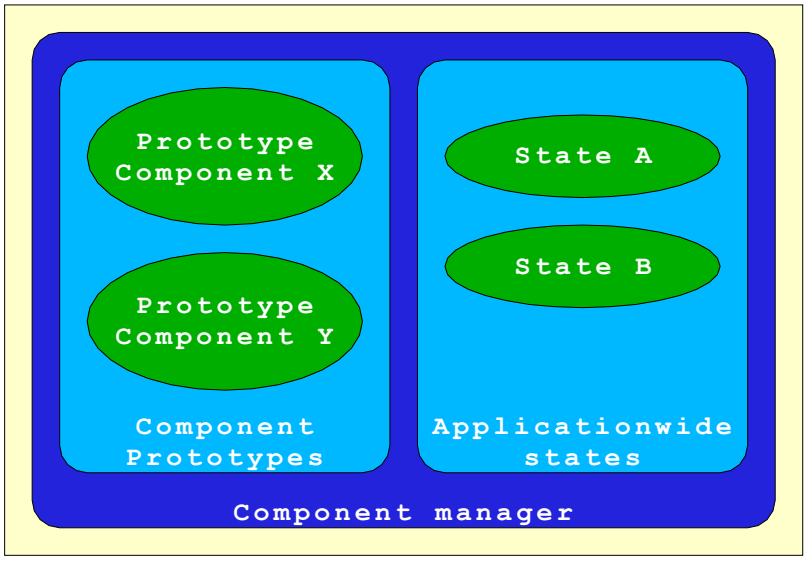

Figure 16: Component manager

6.6. Persistence

Making only components persistent will not make the whole application persistent. Also the connections must be persistent. To accomplish the persistence of components and connections an export and import mechanism for the component manager is required.

6.6.1. Export

A component manager is exported by the component writer (as seen in Figure 17). This is done in three steps:

- Export prototypes of all composed components. Therefore we have to export all composed components that consist of programmed components or exported composed components. This must be done recursively until all component prototypes are exported. This kind of sorting is important for the import. It makes it easier to implement the import mechanism.
- Export all component instances of the component network stored in the component manager by exporting the name of the component prototype and the component configuration property for each component. At the export of the components an export id must be assigned to each component. This id starts with zero and will be incremented for each component.
- Export all connections by exporting the ids of the emitter and receiver components and the names of in- and out-slot for each connection.

Like the property writer the component writer is the abstraction of the different formats like:

- Database
- XML format
- Binary stream format

So the component writer will be an abstract base class providing the interfaces required for the export of the whole component manager. To implement a component writer for a specific format you have to derive from this base class and implement the abstract interfaces.

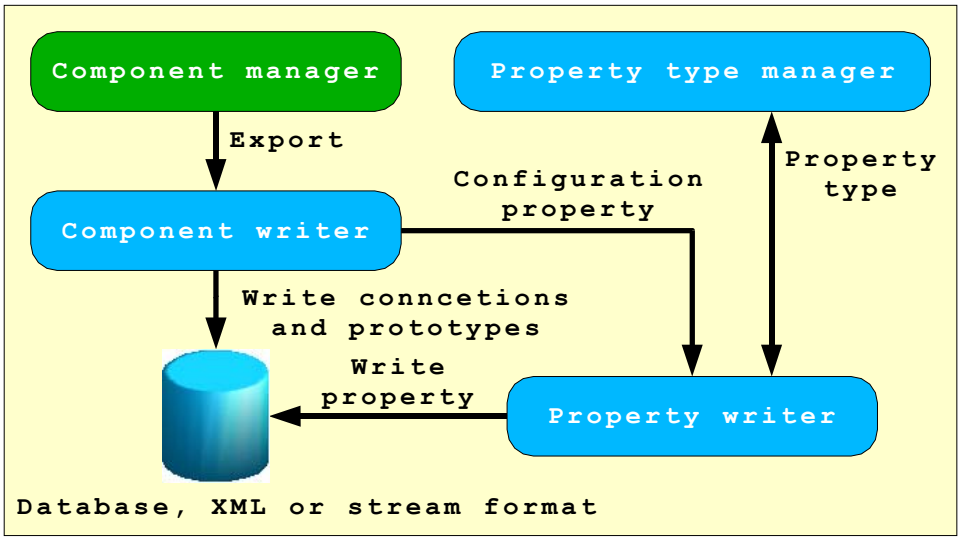

Figure 17: Export of components and connections

6.6.2. Import

A component manager is imported by the component reader (as seen in Figure 18). This is done in three steps:

- Import all composed component prototypes. The component library of the application must register normal components, which are programmed. They can't be exported.
- Import all component instances by importing the name of the component prototype and the component configuration property. For each component a new instance of the component prototype must be created and configured with the imported component configuration property. At the import of the components an import id must be assigned to each component. This id starts with zero and will be incremented for each component.
- Import the connections by importing the emitter and receiver ids and the names of the in- and out-slot. The sequence of component export and import must be the same to ensure that the ids are the same. The evaluation of the component ids and slot names allows the creation of the corresponding connection between the components.

Like the component writer the component reader is also an abstract base class to provide an abstraction of the different formats. An implementation must be provided for each supported format.

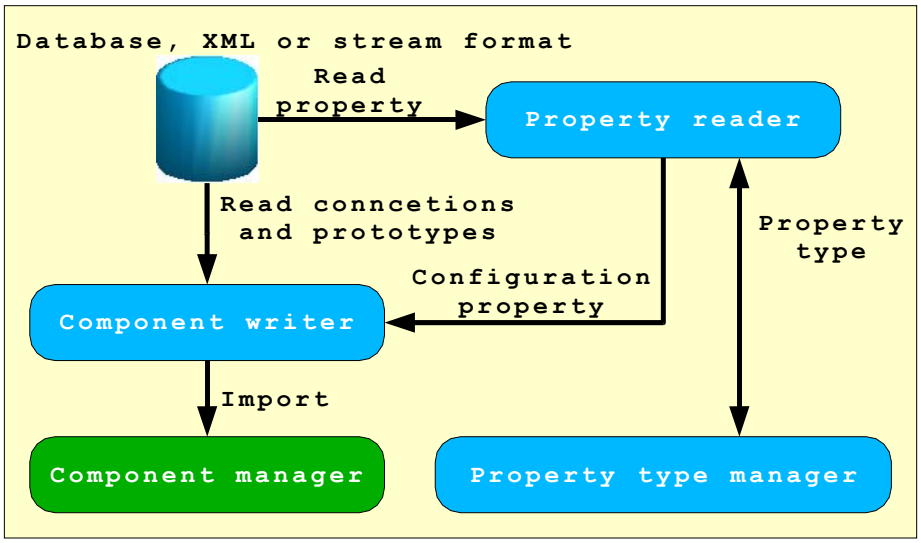

Figure 18: Import of components and conncetions

6.7. Specification

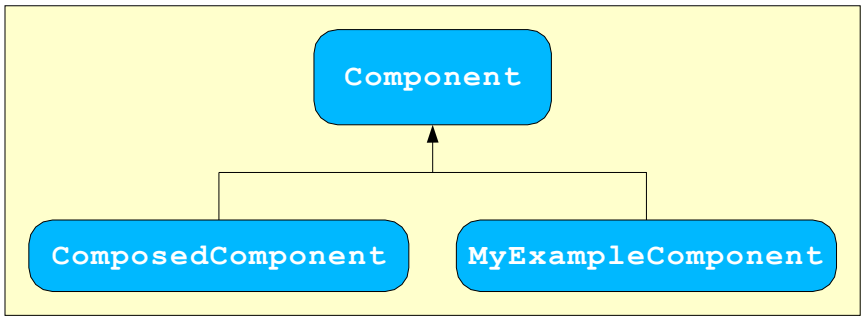

Figure 19: Component hierarchy

In this section we specify the C++ implementation architecture of the components. Therefore we begin with the inheritance hierarchy of the components as seen in Figure 19 and the relationship between the classes of the component architecture as seen in Figure 20 and Figure 21. The base class of each component is the class *Component*. It provides the basic functionality to implement you own component easily. Details of this basic functionality are described in the specification of the interfaces. An component implementation that is provided by the framework is the class *ComposedComponent*. It provides all necessarily functionality to wrap a network of components into a single component and export several in- and out-slots.

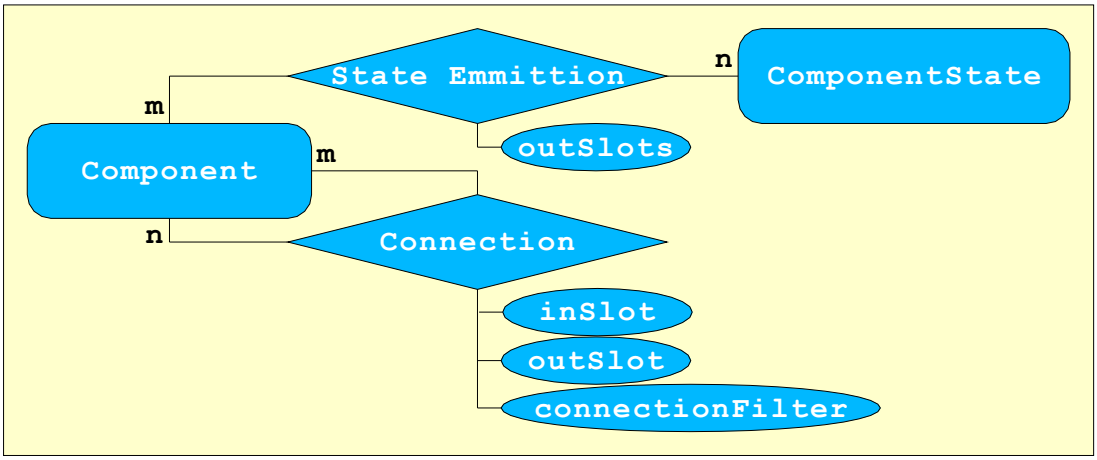

Figure 20: Connection relationships in the component architecture

The relationships in the component architecture that are responsible for the connection of two components are shown in Figure 20.

- A component can be connected to n components and can be the target of m connections. A connection between two components is defined by the emitter, the receiver, the out-slot, the in-slot and a connection filter that enables the framework to implement blocking mechanisms for connections in the future.
- Components are state oriented. Therefore a component provide an interface to manipulate the state and to register out-slots that have to be emitted when the component switches to a spcific state. This is shown in the relationship between the component and the component state, which is also a n to m relations. This means a component can have n registered states and a component state can be registered in m components.

In Figure 21 a small overview of the relationships between the component manager, component prototypes, component instances, GEMs and the property type manager is given. On the lowest level we have the GEMs they provide several basic functionalities to the component experts.

The component expert uses these GEMs to develop component classes that are derived from the class Component or an previously developer component class except the class *ComposedComponent*, which should be extended by a framework expert. This set of classes build the basic component classes level. These classes must exist at the compile time or have to be registered by a dynamic reload of dynamically linked libraries (like ".dll" files in Windows or ".so" files on most UNIX platforms).

The next level is component prototype level. In this level you have to instantiation component prototypes and register them at a component manager. These component prototypes can be instances of native implemented component classes or composed components that should be available for reuse.

The last level is the application level. The component manager and the underlying MR engine are the main parts of this level. The component manger is responsible for the creation of new components out

of the registered component prototypes. Therefore it is important that component prototypes are not directly used in the application to ensure the stability of the prototypes. The component manager also contains his own property type manager to ensure that two component managers are able to have different property types with the same name. Further this separation is important for the component prototypes. Two component manager are able to have different component prototype with the same name. This complete separation allows us to use the component manager as the separation mechanism between two applications. This means you are able to load several different application and switch between them by registering them at the MR engine, which delegates important call-backs to the component manager like the display of a frame.

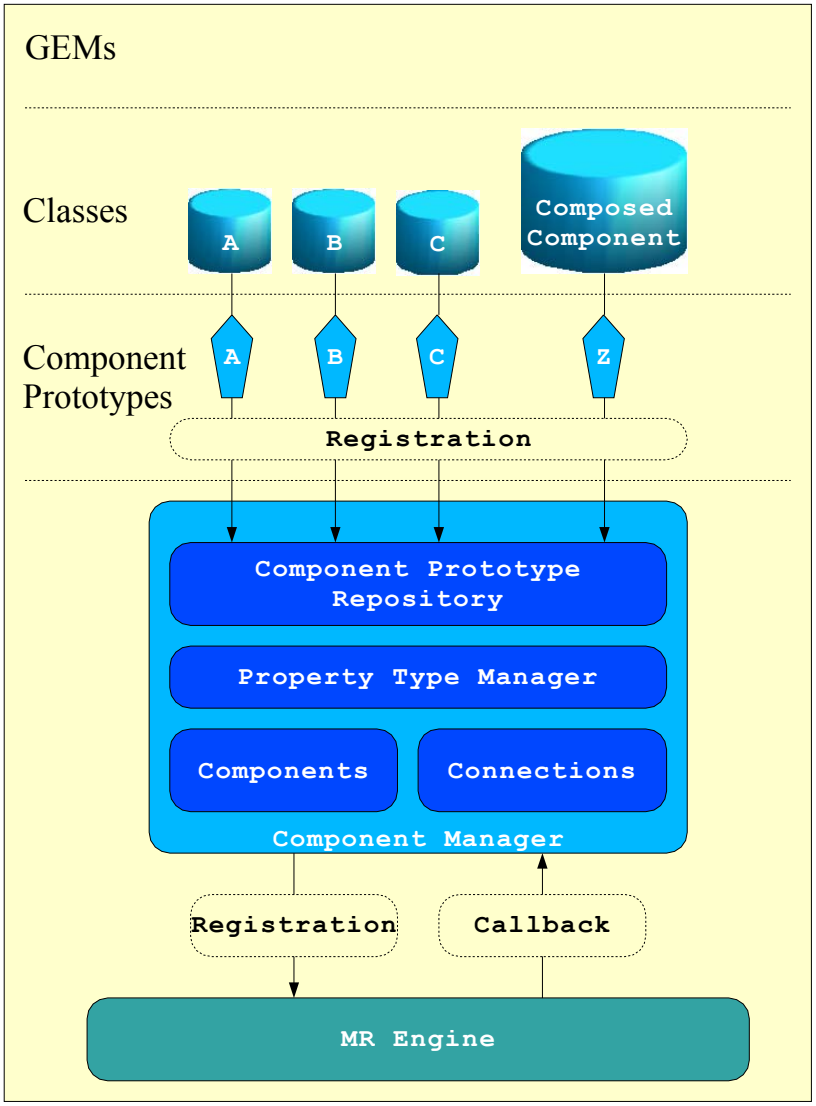

Figure 21: Overview of relations concerning the component manager

6.7.1. Component class

The class *Component* is the abstract base class of all component implementation classes. To implement your own component you have to extends the class *Component* and implement the *clone* method. The integration of slots can be done by implementing the methods *receiveInSlotProperty* and *emitOutSlotProperty* and register the slots by using the methods *registerInSlot* and *registerOutSlot*. The registration will map the slot names to slot ids, which are faster to compare. The implementation of functionality that must be performed for each frame should be placed in the methods *functionalCallback*, *occluderCallback* and *displayCallback*. These methods of all components are called by the MR engine for each frame. First all *functionalCallback* methods are called, then all *occluderCallback* methods and at last all *displayCallback* methods. This means the MR engine starts with the logical and functional iteration, followed by displaying all real occluders preparing the z-buffer for the augmented geometry and the last iteration displays the augmented geometry.

```
class Component {
public:
  Component(ComponentManager *componentManager = NULL);

  virtual ~Component();

  virtual void functionalCallback();
  virtual void occluderCallback();
  virtual void displayCallback();

  virtual bool isConnectable(int outSlotId, Component *receiver,
                             int inSlotId);

  void lock();
  void unlock();

  bool connect(int outSlotId, Component *receiver, int inSlotId,
               const ConnectionFilter *filter = NULL);
  void disconnect(int outSlotId, Component *receiver, int inSlotId);

  const vector<int> &getStateEmittions(const ComponentState *state);
  void setStateEmittions(const ComponentState *state,
                         const vector<int> &outSlotIds);

  void addStateEmittion(const ComponentState *state, int outSlotId);
  void removeStateEmittion(const ComponentState *state, int outSlotId);

  void emit(int outSlotId);

  void setEmitSynchronized(bool synchronized = true);
  bool isEmitSynchronized();

  int numberOfInSlots() const;
  int numberOfOutSlots() const;

  const string &getInSlotName(int inSlotId) const;
```

```
  const string &getOutSlotName(int outSlotId) const;

  const PropertyType *getInSlotType(int inSlotId) const;
  const PropertyType *getOutSlotType(int outSlotId) const;

  int getInSlotId(const string &inSlotName) const;
  int getOutSlotId(const string &outSlotName) const;

  void setState(const ComponentState *state);
  const ComponentState *getState() const;

  const string &getPrototypeName();
  long getId() const;

  const Property *getConfigurationProperty();
  Property *requestConfigurationProperty();
  void releaseConfigurationProperty();

  const vector<vector<Connection *> > &getEmitConnections() const;
  const vector<vector<const Connection *> > &getReceiveConnections() const;

protected:
  virtual Component *clone() = 0;

  virtual void receiveInSlotProperty(int inOutSlotId, Property *property);
  virtual Property *emitOutSlotProperty(int outSlotId);

  virtual void configurationPropertyModified() = 0;

  int registerInSlot(const string &inSlotName,
                     const PropertyType *propertyType);
  int registerOutSlot(const string &outSlotName,
                      const PropertyType *propertyType);

  void setPrototypeName(const string &prototypeName);

  void setConfigurationProperty(Property *property);
 };
```

The '**virtual void functionalCallback()**' method must be overwritten by the derived component implementation to integrate the logical functionality into the component. An assert exception should be thrown in any failure case.

The '**virtual void occluderCallback()**' method must be overwritten by the derived component implementation to display real objects that are able to occlude augmented objects. An assert exception should be thrown in any failure case.

The '**virtual void displayCallback()**' method must be overwritten by the derived component implementation to display augmented objects. An assert exception should be thrown in any failure case.

The '`virtual bool isConnectable(int outSlotId, Component *receiver, int inSlotId)`' method returns true when the connection between the specified out-slot of this component and the specified in-slot of the receiver component is possible. Otherwise false is returned. The default implementation compares the property types. A connection is only possible when the property types are equal. An assert exception is thrown when the specified in- or out-slot is not available or when the given receiver is null.

The '`void lock()`' method locks this component for the current thread. Only one thread is able to get the lock for a component at the same time. The method blocks if the component is already locked until the lock is released by an unlock. They are managed in a waiting queue by the multithreading library. The lock must be used to mark the begin of a program part that is a mutual exclusion area, which means no concurrent processing is allowed and it has to be atomic for the concurrent threads. An assert exception is thrown when any trouble occurs with the locking of the component. Like the detection of a deadlock for example.

The '`void unlock()`' method unlocks this component for the current thread. The Only one thread is able to get the lock for a component at the same time. The next thread in the waiting queue gets the lock and can proceed with the mutual exclusion area. The unlock must be used to mark the end of such a mutual exclusion area. An assert exception is thrown when any trouble occurs with the unlocking of the component.

The '`bool connect(int outSlotId, Component *receiver, int inSlotId, const ConnectionFilter *filter = NULL)`' method connects the specified out-slot of this component with the specified in-slot of the given receiver component and returns true when the connection is possible. Otherwise false is returned. A connection can only be established when the method *isConnectable* returns true. An assert exception is thrown when the specified in- or out-slot is not available, when the given receiver is null or when the *isConnectable* method of the component implementation throws an exception.

The '`void disconnect(int outSlotId, Component *receiver, int inSlotId, const ConnectionFilter *filter = NULL)`' method disconnects the specified out-slot of this component with the specified in-slot of the given receiver componentAn assert exception is thrown when the specified in- or out-slot is not available, when the given receiver is null, or when the specified connection does not exist.

The '`const vector<int> &getStateEmittions(const ComponentState *state)`' method returns a vector containing the ids of all out-slots that are emitted when the state of the component is set to the given state. An assert exception is thrown when the given state is null.

The '`void setStateEmittions(const ComponentState *state, const vector<int> &outSlotIds)`' method replaces the previously defined list out-slots that must be emitted when the component state is set to the given state with the list of out-slot specified in the given vector of out-slot ids. An assert exception is thrown when the given state is null or when a out-slot out of the vector does not exist.

The '`void addStateEmittion(const ComponentState *state, int outSlotId)`' method adds the specified out-slot to the list of out slots that are emitted when the component state is set to the

given state. An assert exception is thrown when the given state is null or when the out-slot does not exist.

The '`void removeStateEmittion(const ComponentState *state, int outSlotId)`' method removes the specified out-slot from the list of out slots that are emitted when the component state is set to the given state. An assert exception is thrown when the given state is null or when the out-slot does not exist.

The '`void emit(int outSlotId)`' method emits the specified out-slot. This method should be used in the *receiveInSlotProperty* or *functionalCallback* method to implement the behavior of the component that is visible to other components. An assert exception is thrown when the overwritten method *emitOutSlotProperty* throws an assert exception.

The '`void setEmitSynchronized(bool synchronized = false)`' method enables or disables the thread synchronization of all out-slot emits for this component. The default setting that all emits are not synchronized. When you set it to true the whole component becomes a mutual exclusion area.

The '`void isEmitSynchronized()`' method returns the synchronization setting for the out-slot emits of this component. True means for each emit the component is handled as a mutual exclusion area. False means emits on this component can be concurrently processed by several threads.

The '`int numberOfInSlots() const`' method returns the number of in-slots that are defined for this component.

The '`int numberOfOutSlots() const`' method returns the number of out-slots that are defined for this component.

The '`const string &getInSlotName(int inSlotId) const`' method maps the given in-slot id to the in-slot name. An assert exception is thrown when the specified in-slot does not exist.

The '`const string &getOutSlotName(int outSlotId) const`' method maps the given out-slot id to the out-slot name. An assert exception is thrown when the specified out-slot does not exist.

The '`int getInSlotId(const string &inSlotName) const`' method maps the given in-slot name to the in-slot id. An assert exception is thrown when the specified in-slot does not exist.

The '`int getOutSlotId(const string &outSlotName) const`' method maps the given out-slot name to the out-slot id. An assert exception is thrown when the specified out-slot does not exist.

The '`void setState(const ComponentState *state)`' method sets the component state to the given state and emits all out-slots that are on the list of out-slots that have to be emitted for this state. An assert exception is thrown when the given state is null.

The '`const ComponentState *getState() const`' method returns the current state of the component.

The '`const string &getPrototypeName()`' method returns the name of the component prototype.

The '`long getId() const`' method returns the id of the component.

The '`const Property *getConfigurationProperty()`' method returns reference pointer of the not modifiable configuration property for this component.

The '`Property *requestConfigurationProperty()`' method locks the component and returns reference pointer of the modifiable configuration property for this component. An assert exception is thrown when the configuration property is requested twice.

The '`void releaseConfigurationProperty()`' method unlocks the component and notifies the component that the configuration property has been modified. An assert exception is thrown when the configuration property has not been requested previously.

The '`const vector<vector<Connection *> > &getEmitConnections() const`' method returns a connection list for all out-slots of this component. The first index specifies the id of the out-slot and addresses the list of its connections.

The '`const vector<vector<const Connection *> > &getReceiveConnections() const`' method returns a connection list for all in-slots of this component. The first index specifies the id of the in-slot and addresses the list of its connections.

The '`virtual Component *clone() = 0`' method must return a clone of this component. It is used by the component manager to create new component instances out of the component prototype. It is abstract and has to be implemented by each component implementation.

The '`virtual void receiveInSlotProperty(int inOutSlotId, Property *property)`' method must be implemented to integrate in-slots into the component implementation. It is called when the component receives a property at the in-slot specified by its in-slot id. An assert exception should be thrown for any failure case.

The '`virtual Property *emitOutSlotProperty(int outSlotId)`' method must be implemented to integrate out-slots into the component implementation. It is called, when the out-slot specified by the out-slot id is emitted by the component itself or from outside the component. It must provide the property that should be produced for this emit. An assert exception should be thrown for any failure case.

The '`virtual void configurationPropertyModified() = 0`' method is an abstract method that must be implemented by the component implementation class. It is called each time when the configuration property is released, which implies the possibility of a modified configuration property.

The '`int registerInSlot(const string &inSlotName, const PropertyType *propertyType)`' method registers the given in-slot name and assigned the given property type for it. It must be used to register the integrated the in-slots into the component implementation. It returns

the in-slot id for the given in-slot name. An assert exception is thrown when the given property type is null.

The '**`int registerOutSlot(const string &outSlotName, const PropertyType *propertyType)`**' method registers the given out-slot name and assigned the given property type for it. It must be used to register the integrated the out-slots into the component implementation. It returns the out-slot id for the given out-slot name. An assert exception is thrown when the given property type is null.

The '**`void setPrototypeName(const string &prototypeName)`**' method sets the name of the component prototype. It should be used in the constructor of the component prototype implementation to specify its name.

The '**`void setConfigurationProperty(Property *property)`**' method sets the configuration property. It should only be used in the constructor of the component. An assert exception is thrown when the given property is null or the configuration property is already set.

6.7.2. ComposedComponent class

An instance of the composed component contains a set of components, a set of connections between the components and a set of exported in- and out-slots. When you have a network of components which does something special that should be reusable you grab this network and wrap it into a composed component. After the registration of this composed component at the component manager it is available for reuse and can be handled like each normal component implementation.

```
class ComposedComponent : public Component {
public:
  class SlotExport {
  public:
    SlotExport(bool inSlot, Component *component, const string &slotName);
    SlotExport(bool inSlot, Component *component, int slotId);

    Component *getComponent() const;
    int getSlotId() const;
    bool isInSlot() const;
    const string &getSlotName() const;
  };

  ComposedComponent(ComponentManager *componentManager = NULL);
  ComposedComponent(const vector<Component *> &components,
                    const vector<SlotExport> &inSlotExports,
                    const vector<SlotExport> &outSlotExports,
                    ComponentManager *componentManager = NULL);

  virtual ~ComposedComponent();

  virtual bool isConnectable(int outSlotId, Component *receiver,
                             int inSlotId);

  void addComponent(Component *component);
  void removeComponent(Component *component);
```

```
  void addSlotExport(const SlotExport &export);
  void removeSlotExport(const SlotExport &export);

  void addComponent(const vector<Component *> &components);
  void removeComponent(const vector<Component *> &components);

  void addSlotExports(const vector<SlotExport> &exports);
  void removeSlotExports(const vector<SlotExport> &exports);

  void eliminateUnusedExports();

protected:
  virtual Component *clone();

  virtual void receiveInSlotProperty(int inOutSlotId, Property *property);
  virtual Property *emitOutSlotProperty(int outSlotId);

  virtual void configurationPropertyModified();
};
```

The '**virtual bool isConnectable(int outSlotId, Component *receiver, int inSlotId)**' method returns true when the connection between the specified out-slot of this component and the specified in-slot of the receiver component is possible. Otherwise false is returned. The default implementation compares the property types. A connection is only possible when the property types are equal. An assert exception is thrown when the specified in- or out-slot is not available or when the given receiver is null.

The '**void addComponent(Component *component)**' method adds a component to the set of components contained in this composed component. Each in- and out-slot connection of the given component must be analyzed. Connections to an in- or out-slot export of this composed component must be connected directly to the involved component inside the composed component. After the integration of the given component into the composed component ist connections to a component in- or out-slot outside of the composed component must be reconnected. Each slot of the given component that is involved in such a connection must be exported and the involved slot of the outside component must be connected to the exported slot. Slot exports that are required before the integration of the given component and are not required after the integration are not removed from the export list. Under the assumption that the export only was include into the export list to uphold the connection between an inside and an outside component the removal of the export would be the correct behavior. But this is only one special case and therefore it cannot be the default behavior. An assert exception is thrown when the component is null or it is already part of the composed component.

The '**void removeComponent(Component *component)**' method removes the component from the set of components contained in this composed component. It is not a deletion of the component. Like in the *addComponent* method it is necessary to update the connections lists for the given component and the and export lists of the composed component. Connections between the given component and a component inside the composed component are directly connected before the removal. After the removal they must be connected by an export of the involved slot. Connections between the given component and a component outside the composed component are connected by an export slot. This connection must be transformed to a direct connection after the removal. Due to the

same reason as in *addComponent* method the slot export that are not required anymore after the removal are not removed from the export lists. An assert exception is thrown when the component is null or it is already part of the composed component.

The '`void addSlotExport(const SlotExport &export)`' method adds a not slot export to the list of exported slots. An assert exception is thrown when the component specified by the slot export is not part of this composed component, the specified slot is not available for the component or the export is already part of the export list.

The '`void removeSlotExport(const SlotExport &export)`' method removes the given slot export from the list of exported slots. An assert exception is thrown when the component specified by the slot export is not part of this composed component, the specified slot is not available for the component or the export is not part of the export list.

The '`void addComponent(const vector<Component *> &components)`' method adds the vector of components to the composed component like it is done for a single component. The special feature of this method is the minimization of the unnecessarily exported slots like it would be done when you call the *addComponent* method for each single component.

The '`void removeComponent(const vector<Component *> &components)`' method removes the vector of components from the composed component like it is done for a single component. The special feature of this method is the minimization of the unnecessarily exported slots like it would be done when you call the *removeComponent* method for each single component.

The '`void addSlotExports(const vector<SlotExport> &exports)`' method adds the vector of slot exports to the list of exported slots like it is done for a single slot export.

The '`void removeSlotExports(const vector<SlotExport> &exports)`' method removes the vector of slot exports from the list of exported slots like it is done for a single slot export.

The '`void eliminateUnusedExports()`' method removes all slot exports from the list of exported slots that are not required. This method should be called after you have finished the creation of a new composed component and before the exact specification of the slots that have to be exported and are not exported automatically.

The '`virtual Component *clone()`' method return a clone of this composed component. It is used the clone methods of each component contained in this composed component and connects the cloned components like the original components.

The '`virtual void receiveInSlotProperty(int inOutSlotId, Property *property)`' method forwards the method call the exported slot. An assert exception is thrown for any failure case of the receiver of the forwarded call.

The '`virtual Property *emitOutSlotProperty(int outSlotId)`' method forwards the method call the exported slot. An assert exception is thrown for any failure case of the receiver of the forwarded call.

The '**`virtual void configurationPropertyModified() = 0`**' method forwards the method call the component contained in the composed component. An assert exception is thrown for any failure case of the receiver of the forwarded call.

6.7.3. ComponentManager class

An instance of the class *ComponentManager* holds the whole application which consists of components and connections. Therefore the component manager requires his own property type manager, which implies a set of property types that are defined for the application. A set of component prototypes have to be provided to the component manager and a set of states that are also managed by the component manager. This set of states enables the comparison of states by comparing the references.

```
class ComponentManager {
public:
  static ComponentManager *getDefaultComponentManager();

  ComponentManager();
  ComponentManager(PropertyTypeManager *propertyTypeManager);

  virtual ~ComponentManager();

  void registerState(const ComponentState *state);
  void registerComponentPrototype(const Component *prototype);

  const ComponentMap &getProtoTypes() const;

  const ComponentState *getState(const string &name) const;

  PropertyTypeManager *getPropertyTypeManager();

  Component *newComponent(const std::string &name);
  void deleteComponent(Component *component);

  const vector<Component *> &getComponents() const;
};
```

The '**`void registerState(const ComponentState *state)`**' method registers the given state at this component manager. An assert exception is thrown when the given state is null or a state with the same name is already registered at this component manager.

The '**`void registerComponentPrototype(const Component *prototype)`**' method registers the given component prototype at this component manager. An assert exception is thrown when the given component prototype is null or a component prototype with the same name is already registered at this component manager.

The '**`const ComponentMap &getProtoTypes() const`**' method returns the registered component prototypes.

The '**const ComponentState *getState(const string &name) const**' method returns the registered state with the given name. An assert exception is thrown when the no state is registered for the given name.

The '**PropertyTypeManager *getPropertyTypeManager()**' method returns the property type manager of the component manager. This property type manager must be used to extend the set of available property types.

The '**Component *newComponent(const std::string &name)**' method creates a new instance of the component by calling the clone method of the component prototype with the given name. An assert exception is thrown when no component property type with the given name is registered at this component manager.

The '**void deleteComponent(Component *component)**' method deletes the given component and all connections in which the given component is involved. In the case of a composed component the deletion will be recursively processed for the components inside the composed component. An assert exception is thrown when the given component is null or not part of this component manager.

The '**const vector<Component *> &getComponents() const**' method returns all components that are managed by the component manager.

6.7.4. ComponentManagerReader class

The class *ComponentManagerReader* is the abstract base interface of all component manager reader implementations.

```
class ComponentManagerReader {
public:
  virtual ~ComponentManagerReader();

  virtual ComponentManager *read() = 0;
};
```

The '**virtual ComponentManager *read() = 0**' method is abstract and has to be implemented by each component manager reader. It reads and returns a component manager. An assert exception is thrown in any failure case.

6.7.5. ComponentManagerWriter class

The class *ComponentManagerWriter* is the abstract base interface of all component manager writer implementations.

```
class ComponentManagerWriter {
public:
  virtual ~ComponentManagerWriter();

  virtual write(const ComponentManager *componentManager) = 0;
};
```

The '`virtual write(const ComponentManager *componentManager) = 0`' method is abstract and has to be implemented by each component manager writer. It writes the given component manager. An assert exception is thrown in any failure case.

7. Development Environment

The development environment must support the developer in his/her work. This means when the the developer uses the development environment he/she must be able to do the following tasks:

- Editing: writing the sources as fast as possible under the aspects of the programming guidelines.
- Create the executables for the target system.
- Management of the sources for several developers, who must work in the same source package. This must include the history of source files and the join of concurrent modified source files. The history of a source file allows to access each development step at any time.
- Detect bugs in the executable and find the corresponding source code for the bug.
- Get an overview of the framework he/she is developing on and find parts of the source code like classes, methods or macros as fast as possible. This also must be possible for frameworks he/she is using to develop the own framework. The developer must also get knowledge about this kind of reused frameworks.
- The documentation of the sources for internal and external use. Internal use means that the information will only be available for developers, who must maintain the sources. External documentation will be used by developers that will build their own frameworks by using this documentation and the framework.

Today most of jobs are facilitated by so called integrated development environments (IDE). Therefore they should provide most of the following functionalities:

- Source-navigation and -editing:
  - Modern IDEs highlight the syntax of the programming language to enables the developer to read and understand the source code faster.
  - Developer often use large framework for there development work or they develop in a large framework. In both cases the developer needs an overview of the framework. Therefore the IDE should provide a class browser including also the support of namespaces. This will reduce the time of search for the information like a interface name or a type of a parameter.
  - Code completion also increases the speed of development like the class browser. The IDE must provide the developer all available names (e.g. class names, method name, …) that are available in the current context of his position in the source file.
- Create the executables:
  - Manage the sources of a project
  - Compiling of single source files
  - Building the application out of the compiled sources.
  - Incremental compilation of modified source files.
  - The integration of different compilers.
- Debugging
  - The compiler that is integrated into the IDE must provide debugging information for debuggers in the compiled sources and the executable.
  - The IDE should provide a debugger that allows the developer to find bugs as fast as possible and allows him/her to stop the application at several breakpoints and show the content of variables.
- Documentation:

- The IDE should contain a large set of documentation about the frameworks that could be used in the development. Further it should be able to integrate documentation of other frameworks.
- Optional IDEs also should integrate a documentation tool that is able to create the framework documentation out of the source code. This documentation for example should contain the interface documentation and the inheritance relationships of all classes occurring in the framework.

For the MR framework we have chosen the Microsoft Visual Studio .NET because it supports most of the above features and we already have licenses for it. Unfortunately the management of the source file history and concurrent modification are not so well supported this IDEs. It only supports the tool Microsoft Visual SourceSafe. We have chosen the source version tool CVS [http://www.cvshome.org/] (CVS is the Concurrent Versions System, an open-source network-transparent version control system) that is not supported by Microsoft Visual Studio .NET. But it provides more benefits as Microsoft Visual SourceSafe. Client-server access method for example lets developers access the latest code from any place with an Internet connection. This is very important for all partners because we must exchange the source over the internet and client tools are available on most platforms.

8. Component Technologies

There are currently four main component technologies:

- CORBA
- COM and DCOM
- EJB and JavaBeans

In the next section we give a small overview of the concepts and our reasons why we must implement our own a lightweight component architecture that is able to integrate these technologies.

8.1. CORBA

The Common Object Request Broker Architecture (CORBA) historically was designed to distribute objects to clients applications. This means an application that is mainly based on CORBA is client-server application. On the one hand we have the object request broker (ORB) that provides access to object implementations as seen in Figure 22 [CORBA specification]. On the other hand we have a client who requests the access to the object interfaces from the ORB.

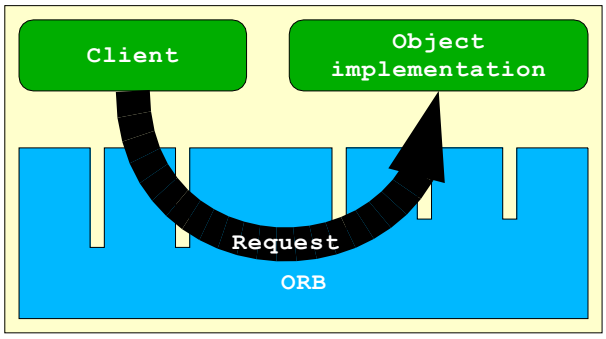

Figure 22: Client access the object implementation via the ORB

On the client side we have in detail several interfaces like the dynamic invocation interface (DII), client stub interfaces and the ORB interface as seen in Figure 23. On the server side we have the skeleton interfaces, the dynamic skeleton interface and the object adapter. Normally most client-server applications are distributed. This means the client send the requests over a network to the server and the server sends the results back to the client also over a network. Therefore the overhead of such an abstraction of the object interfaces compared with the delay time of the network access is irrelevant. In our case we do not have a distributed application that has this client-server architecture. This means we don't have the problem of accessing remote objects and calling methods over a network. In this case the costs of the abstraction layer are important and to high to get a high performance architecture optimized for a local application. This is the first reason why we don't use CORBA to implement the component architecture of the MR framework.

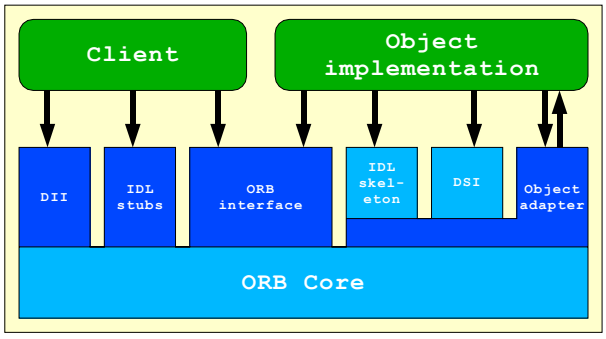

Figure 23: Interfaces of the ORB

The second reason is the development overhead that results form the CORBA concept. As seen in Figure 24 you have to create the interface definition language (IDL) description of your objects. This defines all interfaces for your objects. The IDL compiler creates skeletons for the objects in the target language (e.g. in C++). Additionally stubs are created for the objects in the target language. This stubs are part of the network abstraction and responsible for the method invocation over the network. The skeletons only define that specific interfaces must be implemented but they contain no implementation. You have to implement the interfaces by extending the skeletons. Then you can compile the stubs, skeletons and object implementations. The results must deployed into the server and the client application. One of the goals of the MR component is that they are easy to use and to

develop. A mechanism like this mechanism in CORBA or the RMI stub creation in JAVA is not easy to use for developers. Changing interfaces is always very complicated in such mechanism.

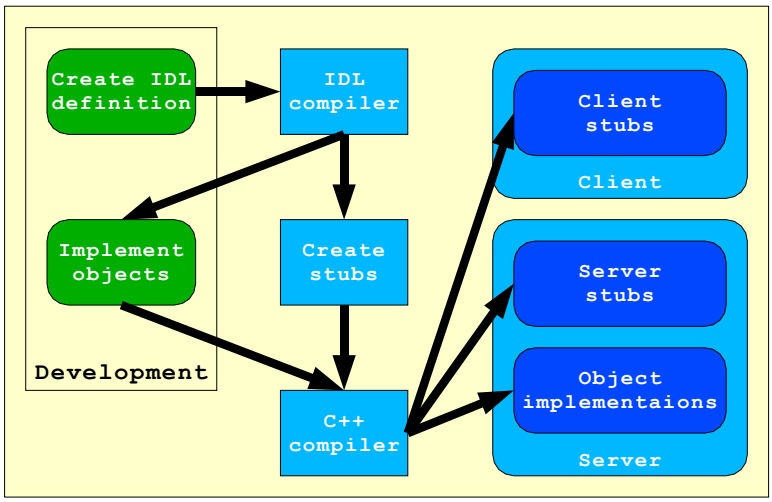

Figure 24: Implementation of CORBA objects

The thirst reason can be found in the object model of CORBA. We have to consider the definition of objects in CORBA. We can find the definition of a object system in the Object Model section of CORBA specification [CORBA specification]. A object system provides services to clients. A client of a service is any entity capable of requesting the service. An object system includes entities known as objects. An object is an identifiable, encapsulated entity that provides one or more services that can be requested by a client. Additionally in the CORBA object model there is no special mechanism for creating or destroying an object for the client's point of view. Objects are created and destroyed as an outcome of issuing requests. This is why the CORBA object model will not fit the requirements of our component architecture in the MR framework. To support a authoring framework a unified mechanism for creating object is required. Further a unified configuration mechanism must exist. In CORBA this possibly could be provided by conventions. A clone method for example, that must be part of each CORBA object. But the problem with conventions is that they are not checked by a compiler. This increases the possibility of bugs in the application.

CORBA provides a lot of nice features for distributed component architectures. It provides

- the registration and lookup of remote object and the network communication between such remote objects.
- a language independent description of object interfaces.
- the creation of stubs and skeletons for several languages. This will allow the integration of components that are written in other languages (e.g. business logic components that are part of an e-commerce framework or a database abstraction layer based on JAVA)
- the network abstraction and remote method invocation.

This features are not required by local components of the AMIRE framework but maybe some future components have to bee distributed. Therefore the MR framework will provide a light weight component architecture that is fast enough to implement real-time MR applications and open enough

to integrate other component technologies as CORBA or DCOM. This will be done with wrapper classes like it has been done for the commercial 3D engine of Folker Schamel. He also has come to the conclusion that CORBA is mainly for distributed components and has developed a micro-component architecture [http://www.usf.de/downl_usf01/Folker_Schamel_-_Einsatz_von_Microkomponenten.pdf] for his 3D engine.

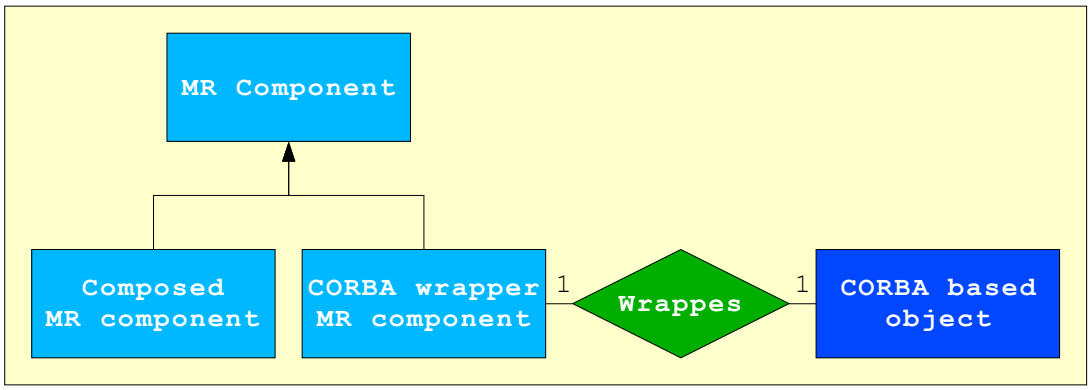

Figure 25: Generic wrapper components over CORBA objects

Especially for CORBA a generic wrapper component class as seen in Figure 25 should be possibly. This can be done by using the IDL definition in combination with DII and DSI of CORBA. These two interfaces will be used to get the reflection data of the object interfaces and invoke the methods by string based names. In Figure 26 the wrapping process of CORBA objects is shown. You request a CORBA object from the remote server and put it into a wrapper component. This wrapper uses the DII and DSI to get the reflection information about the object interfaces. With this information we can create the list of in- and out-slots for the wrapper component. The results and parameters also must be wrapped into and from the CORBA types. When a in-slot gets a property from an outside component the property will be wrapped into a CORBA data type. Then we use the DII to invoke the corresponding method of the stub. This will initiate the remote method invocation of CORBA and the call will be performed on the server. Results invocations are returned to our wrapper component and converted into a property. This will be forwarded to all connected components.

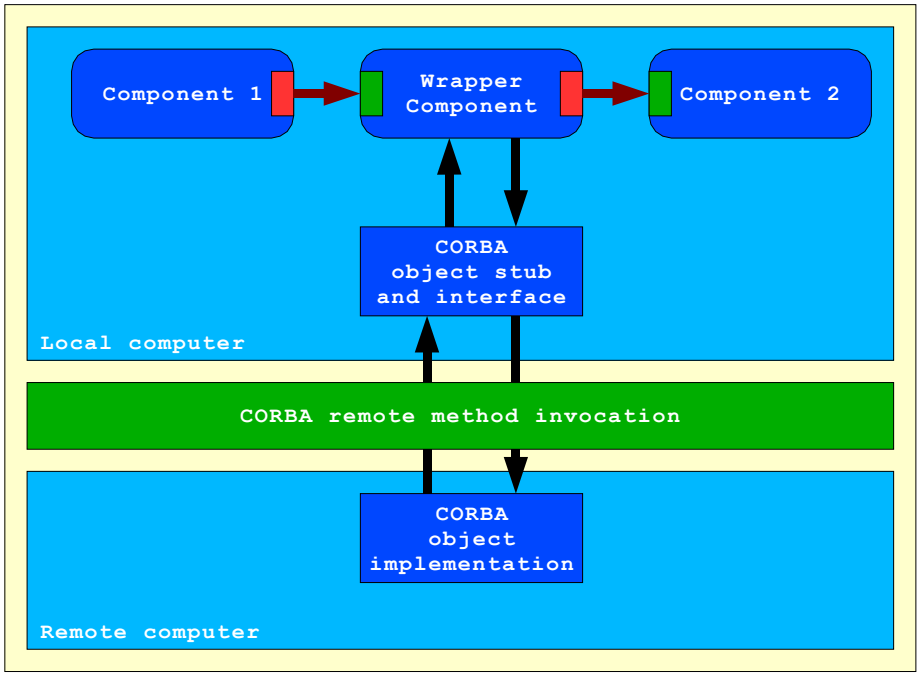

Figure 26: Example of the CORBA wrapping process

8.2. COM and DCOM

We only describe DCOM because it is the distributed extension of COM. Like the description of objects in CORBA the components in DCOM are described with an interface description language. But it is not the same IDL. An important difference between DCOM and CORBA is the dynamic method invocation. In CORBA every object can be handled by the DII and DSI. In DCOM you have to prepare the component by implementing an interface called IDispatch [http://www.execpc.com/~gopalan/misc/compare.html]. Further an universally unique identifier (UUID) called the Interface ID (IID) is assigned to each interface and an UUID called the class identifier is assigned to each class.
DCOM is a Microsoft component standards. This implies that the fully support for this standards is only on a Microsoft system available. DCOM implantations for Linux [Linux DCOM] exist but they are not finished and they are not supported by the origin provider of the standard, who is Microsoft. Therefore the main component architecture of the MR framework should not be implemented with such standards.

8.3. EJB and JavaBeans

JavaBeans is a java based, portable, platform-independent component mode. It enables developers to write reusable components once and run them anywhere. JavaBeans acts as a Bridge between proprietary component models and provides a seamless and powerful means for developers to build components that run in ActiveX container applications [http://www.cetus-

links.org/oo_javabeans.html]. JavaBeans would provide most of the features that are required for our component architecture:

- Persistence
- Properties
- Reflection information on methods and properties (based on the JAVA reflection API).

The EJB concept is also like CORBA or COM a distributed concept. Unfortunately JAVA provides not the performance for a real-time MR application and the EJB concept is not implemented for any other language. Bindings to CORBA are possible over the RMI/IIOP. EJB components are based on the remote method invocation concept (RMI) of JAVA. RMI/IIOP is the integration of the CORBA communication protocol into RMI. Therefore EJB components could be accessed over a CORBA ORB. This allows also the integration into our component architecture by using the CORBA wrapper class.

9. X3D

The goal of the X3D Graphics Working Group is to design the standard for the next-generation Extensible 3D (X3D) Graphics specification [http://www.web3d.org/x3d.html]. The complete X3D specification supports all geometry and behavior description capabilities of the Virtual Reality Modeling Language (VRML 97). The specification is language and encoding independent. A main goal is the encoding using the Extensible Markup Language (XML). It is basically designed for interactive web- and broadcast-based 3D content integrated with multimedia. X3D is intended for use on a variety of hardware devices and in a broad range of application areas such as engineering and scientific visualization, multimedia presentations, entertainment and educational titles, web pages, and shared virtual worlds. X3D is also intended to be a universal interchange format for integrated 3D graphics and multimedia. X3D is the successor to the Virtual Reality Modeling Language (VRML), the original ISO standard for web-based 3D graphics (ISO/IEC 14772-1:1997). X3D improves upon VRML with new features, advanced application programmer interfaces, additional data encoding formats, stricter conformance, and a componentized architecture that allows for a modular approach to supporting the standard.

The X3D standard supports a large list of features of several categories:

- **3D graphics** – Polygonal geometry, parametric geometry, hierarchical transformations, lighting, materials and multi-pass/multi-stage texture mapping
- **2D graphics** – Spatialized text; 2D vector graphics; 2D/3D compositing
- **Animation** – Timers and interpolators to drive continous animations; humanoid animation and morphing
- **Spatialized audio and video** – Audiovisual sources mapped onto geometry in the scene
- **User interaction** – Mouse-based picking and dragging; keyboard input
- **Navigation** – Cameras; user movement within the 3D scene; collision, proximity and visibility detection
- **User-defined objects** – Ability to extend built-in browser functionality by creating user-defined data types
- **Scripting** – Ability to dynamically change the scene via programming and scripting languages
- **Networking** – Ability to compose a single X3D scene out of assets located on a network; hyperlinking of objects to other scenes or assets located on the World Wide Web

- **Physical simulation** – Humanoid animation; geospatial datasets; integration with Distributed Interactive Simulation (DIS) protocols

A lot of this features are also supported by the VRML 97 standard. The problem of VMRL 97 is that it is a monolithic block. This means when you want to support VRML 97 you have to support all features of VRML 97 completely.

X3D splits the specification into components. Each component represents a set of scene node or features. Additionally X3D provides a set of so called profiles. Each profile contains a subset of the X3D components. When you want to support X3D you don't have to support the full X3D standard. It suffices to support one Profile. You have to analyze your application which components of the X3d standard are required by it. Then you can chose the profile that fits best for your application. But you must support all components of this profile to support X3D.

X3D provides five profiles:

- **Interchange** – Supports a minimal subset of X3D needed for import and export of geometry and animations.
- **Interactive** – Supports a minimal subset of X3D needed for interactive content.
- **Extensibility** – Supports a subset of X3D needed for creating extended components and integrated applications.
- **VRML 97 Base** – Supports a subset of X3D corresponding to the full VRML97 feature set and fully compatible with the VRML97 specification.
- **Full** – Supports the complete set of X3D features.

Supporting a component doesn't always require to support the full functionality of the component. The X3D standard provides component support levels to allow the support the component with a subset of its features and nodes. But this means not that you can chose the support level by your own. It is given by the profiles you want to support. The navigation component on support level one for example doesn't support billboards, collision and level of detail. This is only available at support level two, which first is supported by the "VRML97 Base" profile.

The core component of X3D includes the rooting between scene nodes. This is why we have considered to use the X3D standard as the XML export format of the MR components. When we have analyzed the specification we had come to the conclusion that the understanding of MR component and components in the X3D specification is complete different. MR components are more like X3D scene nodes. The X3D standard is currently under construction so therefore no C++ solution for any profile is available. This implies we have to implement at least the interchange profile when we want to use the X3D format as our XML based export format. Which consists of ten components with support level one:

- Core - supplies the base functionality for the X3D runtime system, including the abstract base node type, field types, the event model, and routing.
- Time - includes a definition of the TimeSensor node, the fundamental means for connecting the X3D world to the time base of the browser.
- Aggregation and transformation - includes how nodes are organized into groups to establish a transformation hierarchy for the X3D scene graph.

- Rendering - includes fundamental rendering primitives such as TriangleSets and PointSets, and geometric properties nodes that define how coordinate indices, colors, normals and texture coordinates are specified.
- Geometry - includes how geometry is specified and what shapes are available.
- Shape - defines nodes for associating geometry with their visible properties and the scene environment.
- Lighting - how light sources are defined and positioned as well as how lights effect the rendered image.
- Navigation - defines the nodes that provide support for moving the avatar through an X3D world.
- Interpolation - defines the nodes that provide support for animating various parameters over time through interpolation.
- Texturing - includes how textures are specified and how they are positioned on the subject geometry.

When we analyzed X3D we found several reasons to no support X3D:

- Many parts of the specification are inconsistent with earlier sections of the specification. The name of the rendering component and shape component in the interchange profile for example are called geometric properties component and appearance component: But in the section where all components are described they are introduced as rendering component and shape component.
- A lot of the specification is missing or incomplete. Like the VRML 97 or the binary encoding section of the specification. Some tags of the XML encoding are described by simple examples but not specified. The described tags are only a subset of the data type definition (DTD).
- A large set of internal and external HTML links do not work. Therefore it is hard to get an overview of the whole specification and the external information sources.
- At least we must support all components of the interchange profile to support X3D but we only need the core component for the rooting. This means we only need one of ten components of the standard but we have to implement also the other nine to support X3D. Exporting our MR component network as a X3D file will imply that we also need the extension capabilities of X3D. This means when we want to support X3D we have to support the extensibility profile, which include too much components that are not required by AMIRE.
- The specification of X3D is not complete and therefore also a C++ solution will not be available soon.

This reasons implies that we cannot support the X3D standard for the export format of the MR component network or any MR component. But we will reuse parts of the DTD to create a own XML based export format and DTD. This means we will support a small subset of the X3D standard but we cannot officially support X3D.

10. XML based persistence

The XML based export is specified by in two categories. The first is the export of properties. The second is the export of component managers. The export of component managers reuses the export description of properties. All specifications are based on XML Schema and not XML DTD (data type definition). Because of the missing support for base data types like integer and double in XML DTD.

10.1. Properties

In this section we specify the export of properties as XML Schema. Therefore we begin with the introduction of several simple list types to describe the structure of base vector exports. All values are stored in a XML Schema list. This means a integer vector for example could be stored like this: 1 2 3.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="BooleanList">
    <xs:list itemType="xs:boolean" />
  </xs:simpleType>

  <xs:simpleType name="CharList">
    <xs:list itemType="xs:char" />
  </xs:simpleType>

  <xs:simpleType name="IntegerList">
    <xs:list itemType="xs:integer" />
  </xs:simpleType>

  <xs:simpleType name="FloatList">
    <xs:list itemType="xs:float" />
  </xs:simpleType>

  <xs:simpleType name="DoubleList">
    <xs:list itemType="xs:double" />
  </xs:simpleType>

  <xs:simpleType name="StringList">
    <xs:list itemType="xs:string" />
  </xs:simpleType>
```

The next type describes the structure of a value stored in a structured property type

```
  <xs:complexType name="PropertyValue">
    <xs:sequence minOccurs="1"
                 maxOccurs="1">
      <xs:choice id="PropertyChoise">
        <xs:element ref="BooleanProperty" />
        <xs:element ref="CharProperty" />
        <xs:element ref="IntegerProperty" />
        <xs:element ref="FloatProperty" />
        <xs:element ref="DoubleProperty" />
        <xs:element ref="StringProperty" />
        <xs:element ref="StructProperty" />
        <xs:element ref="CharVectorProperty" />
        <xs:element ref="BooleanVectorProperty" />
        <xs:element ref="IntegerVectorProperty" />
        <xs:element ref="FloatVectorProperty" />
        <xs:element ref="DoubleVectorProperty" />
        <xs:element ref="StringVectorProperty" />
        <xs:element ref="StructVectorProperty" />
      </xs:choice>
    </xs:sequence>
```

```
</xs:complexType>
```

The above referenced elements are defined below. They describe how base property values are stored in a XML file.

```
<xs:element name="BooleanProperty"
            type="xs:boolean" />

<xs:element name="CharProperty"
            type="xs:char" />

<xs:element name="IntegerProperty"
            type="xs:integer" />

<xs:element name="FloatProperty"
            type="xs:float" />

<xs:element name="DoubleProperty"
            type="xs:double" />

<xs:element name="StringProperty"
            type="xs:string" />
```

The element for a structured property value is described below. It consists of the name of the structured property type and a list of field elements. Each field element contains the field name and a property value.

```
<xs:element name="StructProperty">
  <xs:complexType>
    <xs:attribute name="type"
                  type="xs:string" />
    <xs:sequence minOccurs="0"
                 maxOccurs="unbounded">
      <xs:element name="field">
        <xs:complexType>
          <xs:attribute name="name"
                        type="xs:string" />
          <xs:attribute name="value"
                        type="PropertyValue" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The next element defines a structured property type. It contains the name of the structured property type, the name of an optional base type and a list of field definition elements. Each field definiton consists of the field name and a name of the field type.

```
<xs:element name="StructPropertyType">
```

```
    <xs:complexType>
      <xs:attribute name="name"
                    type="xs:string" />
      <xs:attribute name="base"
                    type="xs:string"
                    use="optional" />
      <xs:sequence minOccurs="0"
                   maxOccurs="unbounded">
        <xs:element name="field">
          <xs:complexType>
            <xs:attribute name="name"
                          type="xs:string" />
            <xs:attribute name="type"
                          type="xs:string" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

The next elements describe the base vector property values.

```
  <xs:element name="BooleanVectorProperty"
              type="BooleanList" />

  <xs:element name="CharVectorProperty"
              type="CharList" />

  <xs:element name="IntegerVectorProperty"
              type="IntegerList" />

  <xs:element name="FloatVectorProperty"
              type="FloatList" />

  <xs:element name="DoubleVectorProperty"
              type="DoubleList" />

  <xs:element name="StringVectorProperty"
              type="StringList" />
```

The last two elements describe the root elements of a property export. Containing the structured property types and property values.

```
  <xs:element name="Property">
    <xs:complexType>
      <xs:sequence minOccurs="0"
                   maxOccurs="unbounded">
        <xs:element ref="StructPropertyType" />
      </xs:sequence>
      <xs:sequence minOccurs="1"
                   maxOccurs="1">
        <xs:choice id="PropertyChoise" />
      </xs:sequence>
```

```
    </xs:complexType>
  </xs:element>

  <xs:element name="Properties">
    <xs:complexType>
      <xs:sequence minOccurs="0"
                   maxOccurs="unbounded">
        <xs:element ref="StructPropertyType" />
      </xs:sequence>
      <xs:sequence minOccurs="1"
                   maxOccurs="unbounded">
        <xs:choice id="PropertyChoise" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

10.2. Component manager

In this section we specify the export of component managers as XML Schema. Therefore we begin with the introduction of a string list type and several sequences that are required for the specification.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="properties.xsd" />

  <xs:simpleType name="StringList">
    <xs:list itemType="xs:string" />
  </xs:simpleType>

  <xs:complexType name="ComponentSequence">
    <xs:sequence minOccurs="0"
                 maxOccurs="unbounded">
      <xs:element ref="Component" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="ComposedComponentSequence">
    <xs:sequence minOccurs="0"
                 maxOccurs="unbounded">
      <xs:element ref="ComposedComponent" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="ConnectionSequence">
    <xs:sequence minOccurs="0"
                 maxOccurs="unbounded">
      <xs:element ref="Connection" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="SlotExportSequence">
    <xs:sequence minOccurs="0"
```

```
                maxOccurs="unbounded">
      <xs:element name="SlotExport">
        <xs:complexType>
          <xs:attribute name="componentId"
                        type="xs:string" />
          <xs:attribute name="slotName"
                        type="Property" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
```

The first element in this specification is the connection export element. It contains a component id as string for the emitter and the receiver component plus the in- and out-slot name.

```
<xs:element name="Connection">
  <xs:complexType>
    <xs:attribute name="emitter"
                  type="xs:string" />
    <xs:attribute name="outSlotName"
                  type="xs:string" />
    <xs:attribute name="receiver"
                  type="xs:string" />
    <xs:attribute name="inSlotName"
                  type="xs:string" />
    <xs:attribute name="filterName"
                  type="xs:string" />
  </xs:complexType>
</xs:element>
```

The next element is the export element for emitting states of components. It contains the name of the state and the name of the out-slot that has to be emitted.

```
<xs:element name="EmittingState">
  <xs:complexType>
    <xs:attribute name="stateName"
                  type="xs:string" />
    <xs:attribute name="outSlotNames"
                  type="StringList" />
  </xs:complexType>
</xs:element>
```

The next element is the component export element containing the component id, the configuration property as specified in the previous section (each structured property type is exported only once for a component manager) and the emitting states.

```
<xs:element name="Component">
  <xs:complexType>
    <xs:attribute name="id"
                  type="xs:string" />
    <xs:attribute name="configuration"
                  type="Property" />
    <xs:sequence minOccurs="0"
```

```
                maxOccurs="unbounded">
      <xs:element ref="EmittingState" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The next element is the composed component export element also containing the component id, the configuration property as specified in the previous section (each structured property type is exported only once for a component manager) and the emitting states. Further it contains the inner components and the connections between them plus the export list of in- and out-slots.

```
<xs:element name="ComposedComponent">
  <xs:complexType>
    <xs:attribute name="id"
                  type="xs:string" />
    <xs:attribute name="configuration"
                  type="Property" />
    <xs:sequence minOccurs="0"
                 maxOccurs="unbounded">
      <xs:element ref="EmittingState" />
    </xs:sequence>
    <xs:attribute name="components"
                  type="ComponentSequence" />
    <xs:attribute name="connections"
                  type="ConnectionSequence" />
    <xs:attribute name="inSlotExports"
                  type="SlotExportSequence" />
    <xs:attribute name="outSlotExports"
                  type="SlotExportSequence" />
  </xs:complexType>
</xs:element>
```

The last element is the root element of the component manager export. It is the description of a component manager containing the prototypes, component, composed components and the connections between the components (including the composed components).

```
<xs:element name="ComponentManager">
  <xs:complexType>
    <xs:attribute name="prototypes"
                  type="ComposedComponentSequence" />
    <xs:attribute name="components"
                  type="ComponentSequence" />
    <xs:attribute name="composedComponents"
                  type="ComposedComponentSequence" />
    <xs:attribute name="connections"
                  type="ConnectionSequence" />
  </xs:complexType>
</xs:element>
</xs:schema>
```

11. Main Library Gems

Main library gems provide fundamental MR/AR functionalities. They build the backbone of applications that are developed with the AMIRE framework. Main library gems provide functionality that is more related to the common tasks of MR/AR applications and are therefore more widely used. They are e.g. responsible for racking objects or rendering virtual objects. Other gems would for instance compute vertices that describe the waves that warp the surface of a liquid. The performance of AMIRE based applications is closely related to the performance of those gems. To avoid performance losses, those gems have to be closely integrated into the framework. But this performance gain causes a loss of flexibility. Thus it appears that the number of main library gems must be kept as small as possible. The next section of this chapter contains a description of the main library gem groups and categories that can be identified yet. Then, requirements that a main library gem must fulfill are described, and some implementations that could be used as main library gems and that fulfill those requirements are introduced.

11.1. Main Library Gem Groups and Categories

As aforementioned, main library gems are divided into groups and categories according to their functionality. A group is a more coarse classification, while a category is narrower. E.g. loaders build a group, which is subdivided into the categories image loader, scene loader, video loader etc. The Following describes the two main library groups that can be identified yet. The first one is the loader group, which contains gems that are responsible to load content from a source and transform it to a representation that is appropriate for the framework. The second one is the positioning system group, which includes gems that provide functionality to track the position of objects and/or a camera. Those two main library gem groups represent a good starting point because they cover most of the basic functionality that is needed to develop MR/AR applications. If future development demands additional main library gems, they will be established.

11.1.1. Loader Group

The loader group gems are categorized according to the type of content they load. According to this, there is an image loader category, a scene loader category a sound and a video loader category.

Image Loader Category – Image loaders gems provide functionality to load images from files or a database and to convert them into a format that is suitable for the framework. An image loader gem must be provided for every image format that is supported. To foster exchangeability a plug in mechanism for image loaders will be established. This enforces a unified interface of image loader gems.

Scene Loader Category – Scene Loader gems load the description of a 3d scene and transform it to an appropriate format for the rendering library. The scene's internal representation must also provide the possibility to modify the loaded scene. There must be a scene loader available for every scene description format that should be supported. Scene graph readers are an advancement of scene readers, because they build an object-oriented abstraction of the underlying scene description format and the graphic library. A scene graph reader builds a data structure called scene graph, which describes the scene. This data structure consists of objects, called nodes that describe the structure of the scene. The user of a scene graph does not need to deal with those nodes. The user can address the graph as a whole by calling methods of it's the topmost node. To display such a scene graph, it is sufficient to

call the "display" method of the graph's topmost node. The rest of the display routine, like traversing through every node of the graph in the right order and exhausting the right rendering commands is done by the data structure itself. This greatly alleviates working with 3d scenes. The requirement to enable the manipulation of the loaded scene makes it impossible to build a plug in mechanism similar to the plug in mechanism for image loaders.

Video Loader Category – Video loader gems load video data from a video source like a file containing video data or a video camera. The video data is decompressed if this is necessary and transformed into a format that is suitable for the framework. A video loader gem must at least provide the possibility to stop video playback. A plug in mechanism, similar to that for image loaders will be established. But due to the complexity of video camera interfaces and APIs it has to be evaluated, if such a plug in mechanism can also be used for video cameras.

Sound Loader Category – Sound loader gems load sound data from sound sources like files or microphones. There will be a plug in mechanism built, like that for image loader gems.

11.1.2. Positioning System Group

The gems of the positioning group deliver relative or absolute position and/or orientation data for objects, cameras or persons. This group is divided into the categories optical tracking, magnetic tracking, inertial tracking and positioning systems. Those categories where established according to the needs of the current demonstrator applications. According to the needs of other applications, there can other categories added to this group. Because the interfaces of those gems are relatively uniform, a unified plug in mechanism for all gems of the positioning system group can be established. But this plug in interface will not cover the setup and calibration functions of those gems because they require special procedures, according to the internal functionality of those gems.

Optical Tracking Category – Optical tracking gems base on computer vision technologies. They analyze 2d images, recognize some special items ("markers") and calculate the position and orientation offset between the marker and the camera that created the image. Those systems need no special tracking hardware. The only required hardware is a digital camera and a marker. The marker most times consists of a printout of a special geometric figure like a circle, a rectangle or square. The calibration of those systems consists of a simple camera calibration. This makes those systems very easy to use and cheap. The limitations of those systems are spatial restrictions and the "line of sight" constraint between marker and camera, because the marker has to be clearly identified. But the easy deployment of those systems makes them the first choice for common MR/AR applications.

Magnetic or Ultrasonic Tracking Category – Magnetic or Ultrasonic Tracking systems consist of an emitter that emits magnetic or ultrasonic waves and a receiver that receives those waves. The position and orientation offset between emitter are determined by using physical effects like the runtime of magnetic or ultrasonic waves or signal phase differences. Those systems require special tracking hardware and a relatively costly and complicated setup and calibration process. The accuracy of those systems is higher than the accuracy of optical tracking systems. But those systems are relative stationary because the calibration of those systems has to be repeated for every setup.

Inertial Tracking Category – Inertial Tracking systems base on effects like acceleration and inertia. They consist of micro mechanical devices to measure relative angular and directional movement. Those systems do not depend on an emitter to calculate the relative position and angular offset

between a receiver and an emitter. They measure movements and integrate past movements to calculate the current position and orientation. This causes those systems to "drift". I.e. the measurement error is summed up. Thus, those devices need periodical recalibration.

11.2. Main Library Gem Requirements

Because main library gems are closely coupled to the framework, precautions must be taken to choose suitable implementations for them. To assure this, some requirements for main library gems where established. Those requirements are:

- **Performance –** AMIRE applications shall provide real time performance. This implies that also implementations for main library gem must provide high performance.
- **Quality of Documentation -** The documentation of the implementation must be complete and up-to-date.
- **Public available –** The implementation should be available under LGPL or an equivalent licensing model.
- **Available for Windows Environment –** Most computers run a version of the Windows Operating system and AMIRE should be available for as much MR/AR authors/users as possible. Thus, implementations for main library gems must be available for the Windows environment.
- **Easy to use** – The functionality of a main library gem must be usable easy and intuitive.
- **Good design –** The implementation should be conforming to proper design according to object oriented design guidelines. E.g. it should apply design patterns. It should provide ways to for extending it. The implementation should be flexible and customizable.

Further, some requirements that should be met by main library gems where established:

- **Source code -** To get a deeper and clearer understanding of an implementation it is often better to consult the source code. Thus, it is desired that the source code of the implementation is available.
- **Integrated with Development Environment –** The interfaces, classes or functions of the gem implementation should be built in a way that enables the usage of all features of the development environment, e.g., code completion.
- **Independent of dedicated Hardware –** The gem implementation should not require special hardware, like a special graphic card or a special tracking hardware that can only be supplied by a single vendor.

11.3. Loader Group Gem Candidates

To reduce dependency of different implementation and to benefit from the high level functionality of a scene graph abstraction, it was decided to use a scene graph library as fundamental loader gem. Those scene graph libraries mostly also contain a number of image loaders, which can be used as image loader gems. The following sections contain a short introduction into the principles of scene graphs, followed by a description of two candidate implementations for scene graph gems. The scene loader candidates where selected according to the main library gem requirements described in the previous section. The only two scene graph implementations available for the windows environment that are also available under LGPL license that could be found are OpenSG [OpenSG02] and

OpenSceneGraph [OpenSceneG02]. The gems of the loader categories sound and video are covered by basic multimedia libraries and will not be further described here.

11.3.1. Principles of Scene Graphs

Scene graphs are data structures used to hierarchically organize and manage the contents of 3d scene data. Traditionally considered a high-level data management facility for 3D content, scene graphs are becoming popular as general-purpose mechanisms for managing a variety of media types.

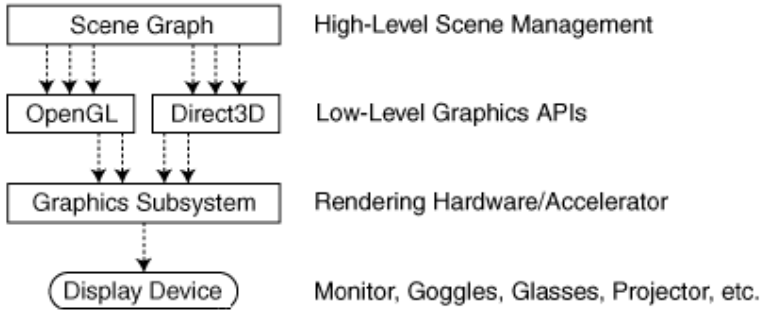

Figure 27: High level scene management with scene graphs

Figure 27 shows one of the main advantages of scene graphs. They provide an abstraction layer to the graphic subsystem responsible for displaying the scene data. Another advance of the scene graph approach is that it provides a clear separation between the scene itself and operations that are performed on the scene.
A scene graph consists of nodes, which are connected by edges. Those nodes and edges build a graph that organizes objects that are represented by nodes hierarchically. There is a special node, called root node at the top of graph. Nodes that contain other nodes are called parent nodes, and nodes that do not contain other nodes are called leaf nodes. The root node is the parent node for the whole graph and has no parent node. Figure 28 shows a generic scene graph.

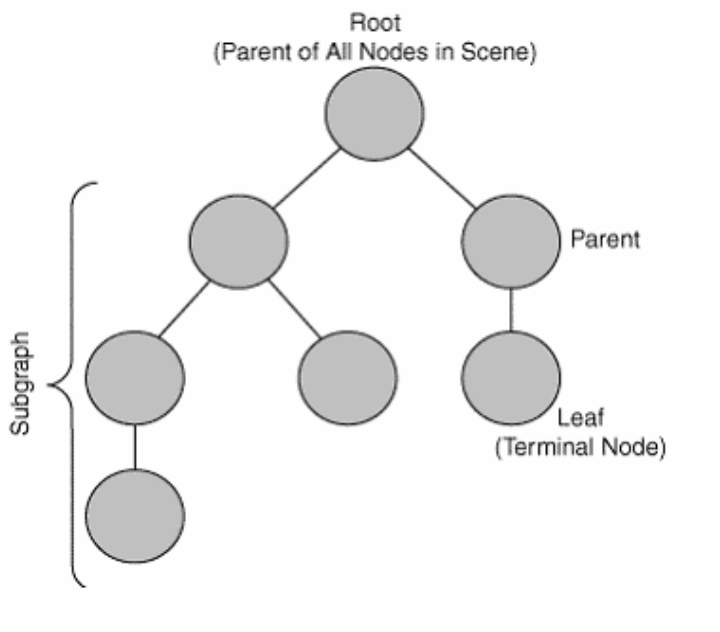

Figure 28: Scene Graph structure

An important functionality of scene graphs is the graph traversal. Graph traversal means that starting from a node, all other reachable nodes will be visited and a certain operation will be performed at every node. Thus, to render the whole scene it is sufficient to call the render method of the root node. This starts traversing from the root node and performs a render operation on every reachable node automatically.

11.3.2. OpenSG

OpenSG is a scene-graph library based on OpenGL [www.opensg.org]. It supports VRML97, OBJ, OFF and RAW formats. The library contains several different nodes to describe 3d content. Besides standard geometry, material and light nodes it also offers distance LOD (level of detail) nodes, billboard nodes and particle system nodes. The whole library is designed in a way to provide multithreaded asynchronous manipulation of the scene graph. To improve performance, the library provides view volume culling and state sorting. The library is available under LGPL and runs on Irix, Linux and Windows systems. Figure 29 shows the basic structure of OpenSG and how it is embedded into other external libraries. The sub libraries `WindowFOXLib`, `WindowQTLib`, `WindowGLUTLib`, `WindowWIN32Lib` and `WindowXLib` contain connections to several windowing systems. The `BaseLib` contains functionality for logging, basic geometry datatypes like matrices and vertices. Multithreading safety and Nodes are implemented in the `SystemLib`.
The supported image formats are jpg, tiff and png. A plug in mechanism allows it to integrate additional image loaders.

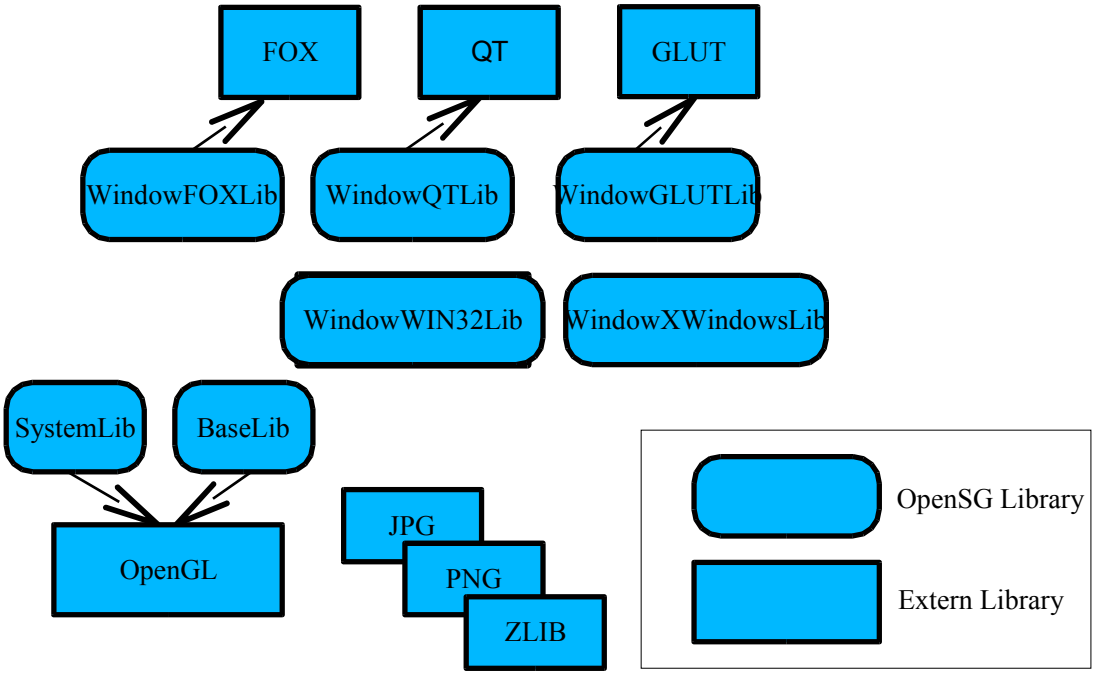

Figure 29: Basic Structure of OpenSG

`Fields` and `FieldContainers`

`Fields` and `FieldContainers` build a basic Concept of OpenSG. A `Field` is a typed data container that holds data that describes a `FieldContainer`. This concept is used to realize a sort of reflectivity in C++. This feature is used to provide multi thread safety. Every thread gets its own "working set" of data. This copy can be manipulated by the thread and will later be merged with the original data.

`Nodes` and `NodesCores`

OpenSG splits the functionality of nodes into the two parts `Node` and `NodeCore`. A `Node` holds information about its position within the scene graph (e.g. a pointer to its parent and pointers to its child nodes) a bounding volume and pointer it's `NodeCore`. A `Node` cannot be shard, i.e. a `Node` can only be at one place in the graph.

A `NodeCore` has a dedicated functionality and contains data according to this functionality. There are `NodeCores` for several different functions. Figure 30 shows an overview of available `NodeCores`. The classes ending with "base", which are derived from `NodeCore` and are base classes for a `NodeCore`, are classes that are automatically generated by a tool called "fcdEdit". This tool takes a XML description of the `Fields` that a `NodeCore` should contain and generates a class that provides methods to manage those `Fields`.

The reason for the concept of `Nodes` and `NodeCores` is to enable the sharing of `NodeCores`. Figure 31 shows an example of shared `NodeCores`. A car has an engine, a body and wheels. Those wheels are mounted at different points. This is modeled by four different wheel transformations. Every wheel transformation has one child node called wheel geometry. Those wheel geometry nodes share the same `NodeCore` containing the data of the wheel geometry.

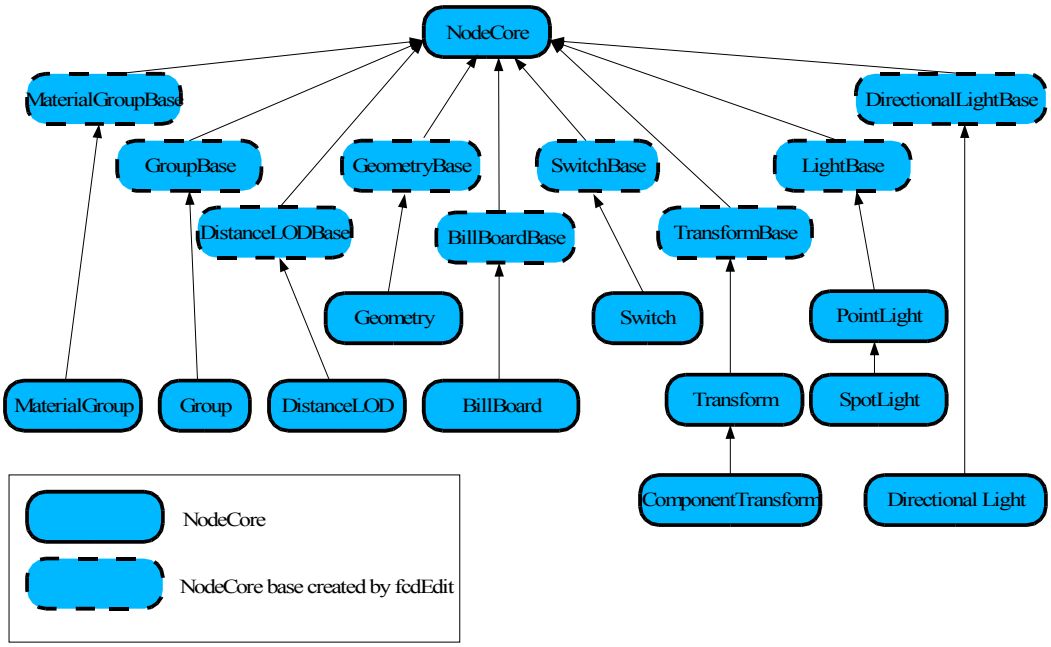

Figure 30: `NodeCores` of OpenSG

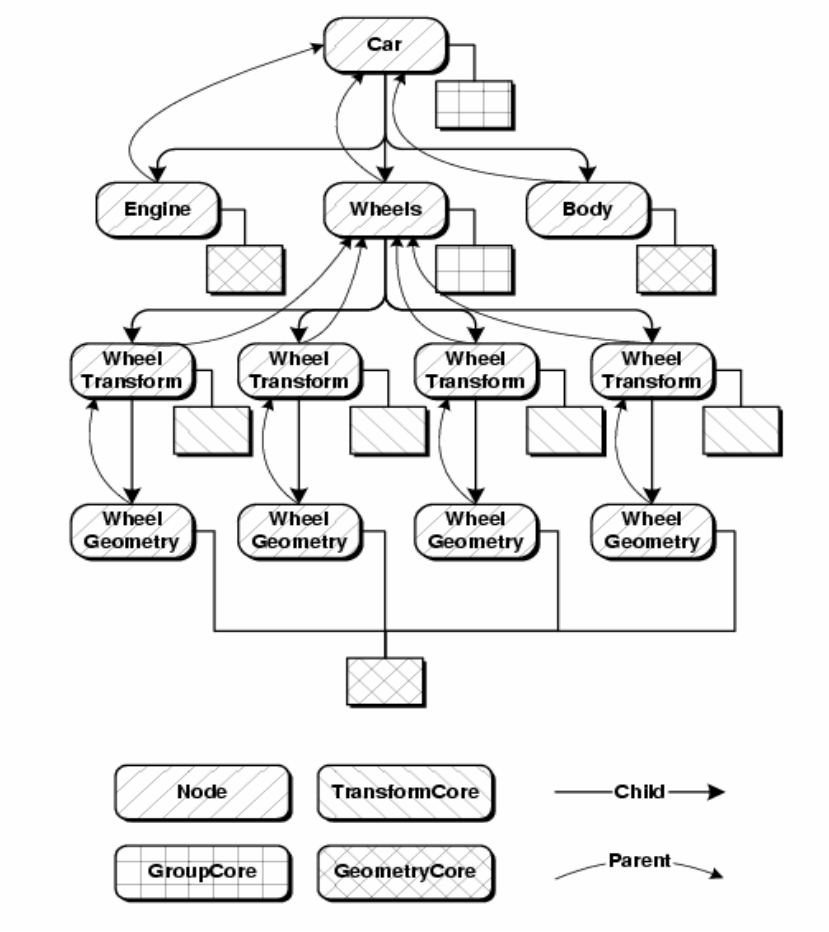

Figure 31: `NodeCore` Sharing

Actions

OpenSG provides two scene traversal operations, called `RenderAction` and `IntersectAction`. The `RenderAction` performs the rendering of a graph. The `RenderAction` does view volume culling and state sorting to improve performance. It handles transparent object by rendering them at last and sorted back to front. The `IntersectAction` takes a line as parameter and tests which of the scene's object are intersected by this line.

11.3.3. Open Scene Graph

Like OpenSG Open SceneGraph is also a scene-graph library based on OpenGL. With several features that are specified on the homepage of Open SceneGraph [www.openscenegraph.org]. It also supports several 3D geometry formats (including the popular 3DS format) and 2d image formats. It is available under the LGPL license for several platforms: Windows, Linux, FreeBSD, IRIX, Mac OSX, Solaris.& HP-UX. The main goals of the Open SceneGraph architecture are:

- Good practices in software engineering
- High performance and quality graphics

- Portability
- Interoperability
- Extensibility

Like the OpenSG library it provides plug-in mechanism to integrate new scene node loader and image loader. The scene graph itself is very similar to OpenGL Performer or Java3D. Programmers that are familiar with one of these frameworks would be easily learn how to develop Open SceneGraph applications.

TO BE DONE

11.4. Tracking Group Gem Candidates

Because the only tracking systems that do not need a special hardware, only systems of this category where considered as tracking gem candidates. The only optical tracking system that is currently available for the windows environment and also available under the LGPL license is the ARToolkit [BlaBil99]. TRIP [deIp02], an optical tracking system developed at the University of Cambridge will probably be available at the end of September. The TRIP system was also considered because, according to the documentation, it should provide better accuracy and range than ARToolkit. An alternative to complete optical tracking systems would be the computer vision library OpenCV. This library does not provide the functionality of an optical tracking system, but it contains all the functionality that is needed to build such a system. The functionality of this library is provided at a relatively high level.
This section contains a short introduction into the principles of optical tracking, an introduction of ARToolkit and TRIP and an overview over the content of OpenCV.

11.4.1. Principles of optical tracking

Optical tracking consists of several tasks that have to be performed. First, an image has to be grabbed from the camera. Then the color or grayscale image has to be converted into a binary black and white image to alleviate further processing of the image. An edge detection algorithm then analyzes the black and white image. This algorithm determines, which areas of the images are edges, e.g. transitions from a black are to a white area or vice versa. Then, the detected edges are analyzed and connected to build shapes. Each shape is analyzed and tested whether it is the shape of a marker or not. It has to be considered that the marker is distorted due to angular offset between camera and marker. If a shape is identified as marker, the distortion of the marker and knowledge of the real shape and size of the marker are used to calculate the distance and angular offset between the camera and the marker. Algorithms that calculate location and orientation of an object or marker from a 2d image are also called POSE or POSIT algorithms. The process and its intermediate images are depicted in Figure 32.

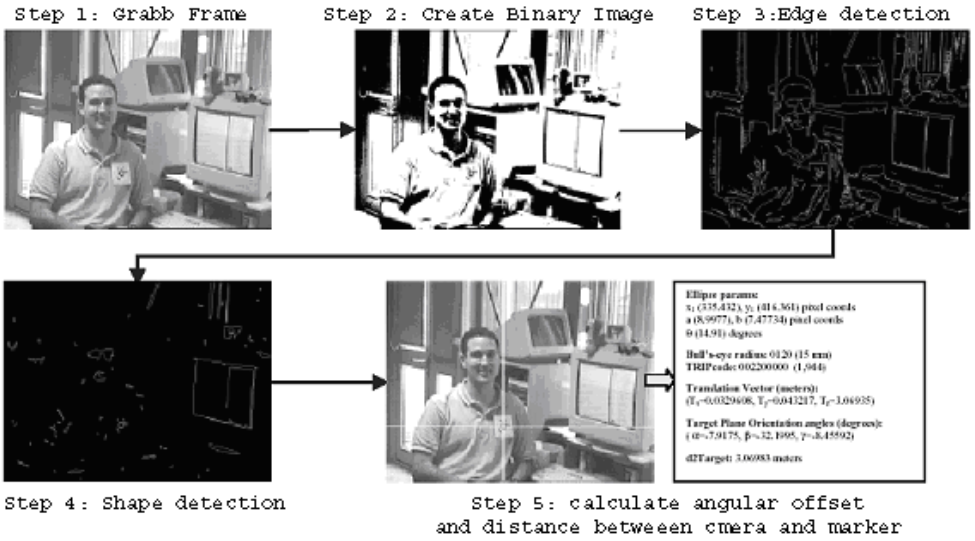

Figure 32: Image processing chain of optical tracking systems

Optical tracking systems depend strongly on the characteristics of the used digital camera. Those characteristics can be described by two sets of parameters, the intrinsic parameters and the extrinsic parameters.

The intrinsic camera parameters of a camera describe the projection that camera performs and the distortion of the camera. The intrinsic camera parameters of a pinhole camera consist of the focal length of the camera, the transformation between the camera frame coordinates and the pixel coordinates and the geometric distortion by the optic. The model of the pinhole camera is depicted in Figure 33. It consists of an image plane Π and a focus point O. The line through O and the center Π is the optical axis and the distance between O and the intersection between the optical axis and Π is the focal length.

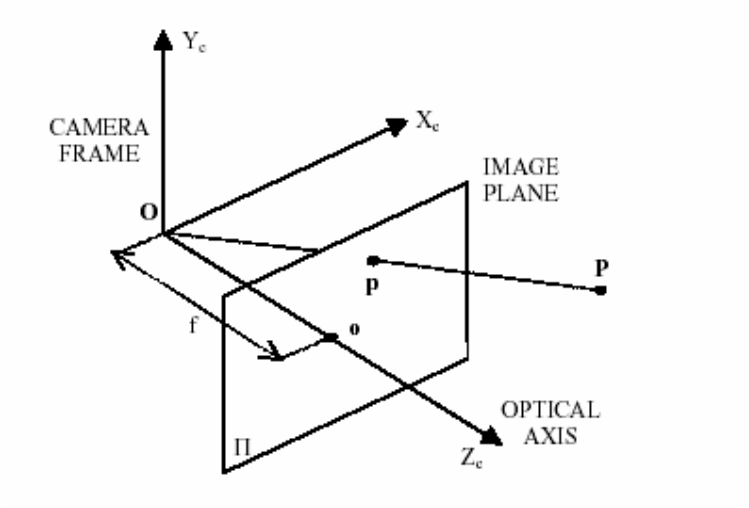

Figure 33 : Pinhole Camera

The transformation between the camera frame coordinates and pixel coordinates depend on the shape and size of the pixels of the CCD chip of the camera. Figure 34 shows how this transformation can be calculated. The coordinates (u,v) of an image point in pixel units must be linked with the coordinates (x_c,y_c) of the same point in the camera reference frame. There may also be a skew between the pixel axes (α), which also has to be considered.

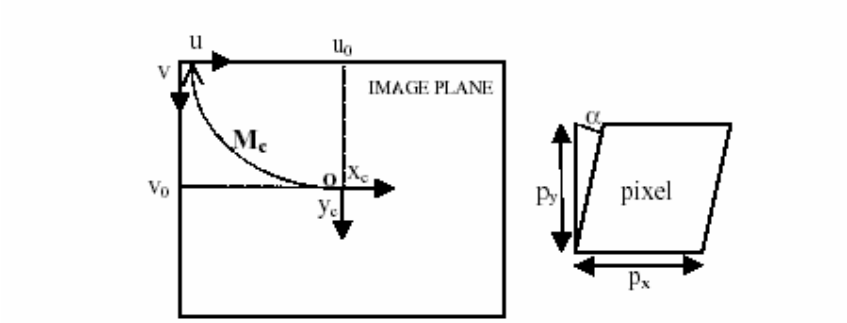

Figure 34 : Transformation from camera frame coordinates to pixel coordinates

The intrinsic parameters of a camera are acquired during a camera calibration process and remain unchanged until the optical characteristic of the camera are changed, e. g. the focus of the camera or the zooming factor is changed.

The extrinsic camera parameters describe orientation and position of the camera related to a rigid reference coordinate system. Assuming that a detected marker determines the reference coordinate system, the extrinsic camera parameters are the distance and angular offset between camera and marker, which is calculated by the optical tracking system. Figure 35 shows the meaning of the extrinsic camera parameters. T is the translation of the reference system's origin and R is the rotation of the axes of reference the system.

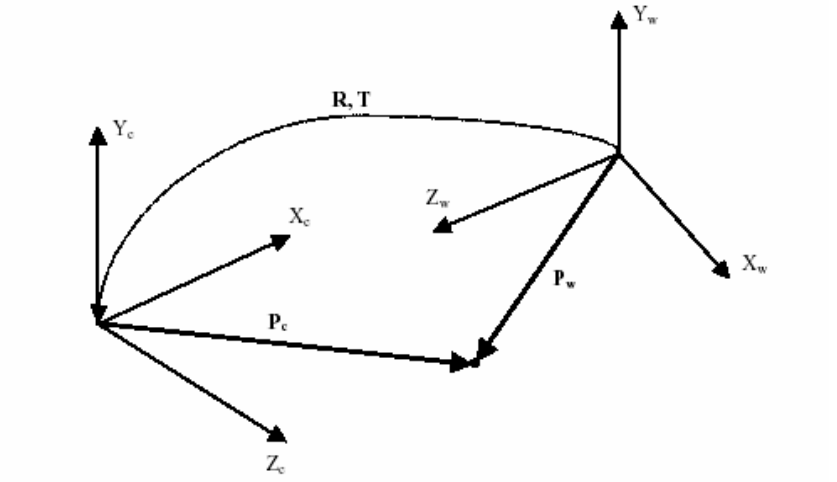

Figure 35 : Extrinsic camera parameters

11.4.2. ARToolkit

The ARToolkit package consists in the tracking system software, a camera calibration tool and a tool called “mk_pattern”. The “mk_pattern” tool is used to teach new markers to the system. This teaching process is needed because ARToolkit uses pattern matching algorithms to identify a marker. A marker consists of two areas. One area is the black thick frame of the marker. This part of the marker is used to detect the marker and to calculate the distance and partly the angular offset between. The angular offset cannot be determined by using information about the frame alone, because the frame is rotationally symmetric. To determine the angular offset, also the second part of the marker, the pattern is needed. The pattern is the area of the marker that is within the frame. The pattern is also used to identify the marker. Figure 36 shows an ARToolkit marker with a pattern that shows the text “Hiro”. The usage of pattern recognition methods to identify markers allows it to use user friendly, more readable markers.

Figure 36 : ARToolkit Marker

11.4.3. TRIP

TRIP (Target Recognition using Image Processing) is a system similar to ARToolkit. It uses circular markers and uses a ternary code called TRIPCode to store information about the size and the identity of the marker within its image. There is no need to teach new markers to the system because size and identity of the markers are coded into the image of the markers in computer readable way. This makes the setup of this system easier, but the markers are less readable than the markers of ARToolkit. Figure 37 shows a TRIP marker. The central dot, the circle and the sync sector are used to determine the distance and angular offset between camera and marker. The two outer circles are used to store the radius and the identity of the marker. The TRIP system is currently not public available. But the author of TRIP was contacted by an AMIRE member and he told that the system will be available at the end of September 2002.

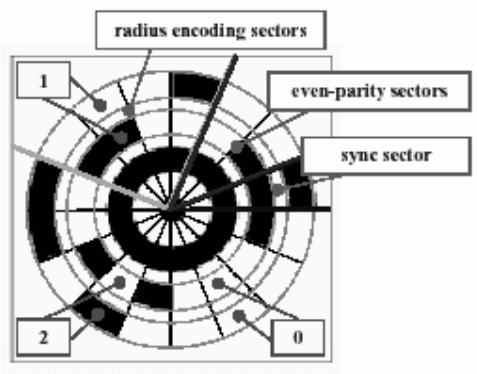

Figure 37 : TRIP marker

11.4.4. OpenCV

OpenCV is an open source video and image processing library patronized by Intel. It does not contain a function that performs optical tracking, but it contains the functionality needed to build an optical tracking system. It provides functionality to binarize images, to find contours and it also contains a POSIT algorithm. Thus, it should be possible to build an optical tracking system on top of OpenCV without excessive efforts. If OpenCV would be used to build a optical tracking system then the functionality provided by OpenCV can be considered as low level gems which are used to build a complex gem that perform optical tracking. This can be done if the available optical tracking systems do not fulfill the demands of the demonstrator applications. Table 1 contains an overview of the functionality of OpenCV.

AREA	Functions
Image functions	Creation, allocation, destruction of images. Fast pixel access macros.
Data Structures	Static types and dynamic storage.
Contour Processing	Finding, displaying, manipulation, and simplification of image contours.
Geometry	Line and ellipse fitting. Convex hull. Contour analysis.
Features	1st & 2nd Image Derivatives. Lines: Canny, Hough. Corners: Finding, tracking.
Image Statistics	In region of interest: Count, Mean, STD, Min, Max, Norm, Moments, Hu Moments.
Image Pyramids	Power of 2. Color/texture segmentation.
Morphology	Erode, dilate, open, close. Gradient, top-hat, black-hat.
Background Differencing	Accumulate images and squared images. Running averages.
Distance Transform	Distance Transform
Thresholding	Binary, inverse binary, truncated, to zero, to zero inverse.
Flood Fill	4 and 8 connected
Camera Calibration	Intrinsic and extrinsic, Rodrigues, un-distortion, Finding checkerboard calibration pattern
View Morphing	8 point algorithm, Epipolar alignment of images
Motion Templates	Overlaying silhouettes: motion history image, gradient and weighted global motion.
CAMSHIFT	Mean shift algorithm and variant
Active Contours	Snakes
Optical Flow	HS, L-K, BM and L-K in pyramid.
Estimators	Kalman and Condensation.
POSIT	6DOF model based estimate from 1 2D view.
Histogram (recognition)	Manipulation, comparison, backprojection. Earth Mover's Distance (EMD).
Gesture Recognition	Stereo based: Finding hand, hand mask. Image homography, bounding box.
Matrix	Matrix Math: SVD, inverse, cross-product, Mahalanobis, eigen values and vectors. Perspective projection.
Eigen Objects	Calc Cov Matrix, Calc Eigen objects, decomp. coeffs. Decomposition and projection.
embedded HMMs	Create, destroy, observation vectors, DCT, Viterbi Segmentation, training and test.
Drawing Primatives	Line, rectangle, circle, ellipse, polygon. Text on images.
System Functions	Load optimized code. Get processor info.
Utility	Abs difference. Template matching. Pixel order<->Plane order. Convert Scale. Sampling lines. Bi-linear interpolation. ArcTan, sqrt, inv-sqrt, reciprocal. CartToPolar, Exp, Log. Random numbs. Set image. K-Means.

Table 1 : Functionality of OpenCV

12. Feasibility Studies

The facility studies where initiated to gather detailed information about the available main library gem candidates OpenSG, OpenSceneGraph and ARToolkit. The subject of this facility study was to find out, whether the two scene graph libraries can be integrated with ARToolkit. While doing this integration, attention was also turned on the requirement conformance of the gem candidates. That is, performance, documentation quality, usability, quality of software design and the integratability with development environment of the gems where also considered during the studies. Because the selection of a scene graph library is essential for the framework, the emphasis of the evaluation was set on those libraries.

12.1. Tasks

There where four task identified, that have to be completed for the integration of an optical tracking system like ARToolkit and a scene graph library:

- **Initialising the Camera –** ARToolKit delivers a matrix that describes the intrinsic parameters of the video camera. This matrix has to be applied to the projection matrix of the virtual camera.
- **Transforming the Object –** The transformation matrix that describes distance and orientation offset between a marker and the camera must be applied to the virtual object.
- **Displaying the Video –** The video image has to be applied to the rendered image as background.
- **"Real Occluders" –** It is often desired that real objects should occlude virtual ones to get a more realistic application. To enable this, rudimentary models of the real environment have to be built. These models can be used to modify the depth information of the rendering system to generate occlusion effects.

12.2. Integration of ARToolkit and OpenSG

12.2.1. Solution

OpenSG is a library that provides multi-threading safety. This makes it easy to use this library in a multi-thread environment. But it also complicates it to extend and modify this library. Every OpenSG node class is derived from a FieldContainer class and has one or more Fields. This Fields contain all data of the Node. To build a new OpenSG node the Fields of this node have to be described with a tool called fscEdit. This tool takes a XML description of a FieldContainer and the Fields it should contain to generate source code which manages the Fields of the node. This generated source code has to be combined with the source code that represents the functionality of the node. Thus, it makes it very hard to extend OpenSG. For this reason we had to try to integrate ARToolKit into OpenSG without extending OpenSG. The first integration task, "initialising the camera" was fairly easy, because OpenSG provides a class called MatrixCamera. This MatrixCamera contains two Matrix Fields, one for the projection matrix and one for the modelview matrix. Thus, setting the projection matrix of a MatrixcCamera with the projection matrix of ARToolKit initialised the OpenSG camera. The next task, "transforming the object" was done by inserting a transformation node at the root of the scene graph and setting it with the modelview matrix of ARToolKit. Displaying the video was a little bit more difficult, because this necessitated an extension of OpenSG. OpenSG uses so called background objects to clear the screen before rendering. For displaying the video we had to create a new

background class that did not only clears the screen, but also displays the video data. We called that background "VideoWallBackground". Due to the fact that a background object does not need own fields, the implementation of this class could be done without using fscEdit. The realization of the "real occluders" was done by first rendering the occluders, then clearing the screen with the VideoWallBackground, (this overwrote the screen with the videoimage, but preserved the depth information) and finally drawing the virtual object.

12.2.2. Evaluation

OpenSG is a powerful tool that supports nearly every feature that is expected from a scene graph. One of the few features that are missing are imposters, a more advanced type of billboards. But the software design of OpenSG is very complex, due to its multi threading safety architecture. This makes it hard to understand the code of OpenSG, which in turn leads to problems if it is affordable to not only use OpenSG as a whole but to split it up and only use parts of it. This complex software design makes it also difficult to create extensions of OpenSG. As mentioned earlier, extensions of OpenSG can only be implemented by using the tool fsgEdit.
The performance of OpenSG is sufficient, but there are performance losses through the multi threading overhead.
The documentation of OpenSG consists of a document called "starter guide" which contains a brief overview over the functionality of the library, a "design document" that summarizes the design decisions drawn during the development of OpenSG, an html-based browsable documentation of OpenSG's classes, several examples and a mailing list. The current documentation of OpenSG is not very detailed and out dated. All the examples are based on a class called SimpleSceneManager which hides away the complexity of interacting with OpenSG. This makes it easy to understand the examples, but undermines the purpose of those examples, to show how to use OpenSG.

12.3. Integration of ARToolkit and OpenSceneGraph

12.3.1. Solution

We have extended the classes GLUT viewer class to integrate the video capturing, object tracking and overlaying into the display method of a MR viewer like it is done in OpenGL. To provide occluding real objects we implemented a MR scene view extending the scene view class of Open SceneGraph. First we display the occluding geometry of the real objects, then we overwrite the color buffer with the video image without manipulating the depth buffer and at last we display the objects recognized by our ARToolKit based object-tracker.

12.3.2. Evaluation

Open SceneGraph is much easier to extend than OpenSG, because OpenSceneGraph does not support multi thread save. Open SceneGraph is also very similar to OpenGL Performer [Perf02], which is a well-known high performance 3D rendering toolkit for developers of real-time, multiprocessed, interactive graphics applications. This similarity makes it much easier to understand how Open SceneGraph works. In combination with an integrated development environment that supports code completion and the Open SceneGraph documentation, it is easy to get an overview of the whole framework and its architecture.

12.4. Conclusion

	OpenSG	**Open SceneGraph**
Easy to understand	OpenSG is a very large and nor so well documented library. It uses a very uncommon type of scene-graph compared with OpenGL Performer [www.sgi.com] or Java3D [java.sun.com]. The scene-graph for example exists of nodes and a node has a node core. On the one hand the node core provides the features like groups or billboards. On the other hand also a node provides group like functionality because it is able to contain child nodes. Therefore it is not easy to understand.	Open SceneGraph is also a very large but better documented library. It is very likely to OpenGL Performer. Therefore it is easy to understand for users of libraries like OpenGL Performer.
Easy to extend	OpenSG provides thread safety by a mechanism that creates copies of the locked object. Therefore a tool is recommended to create new subclasses of any OpenSG class. This tool uses the QT library and is therefore has therefore to be build on windows platforms with a commercial license of QT.	The Open SceneGraph does not use such complex mechanism and is therefore easy to extend.
Supported file formats	OpenSG supports several 3D and picture file formats but the 3DS format was missing	Open SceneGraph also supports several 3D and picture file formats including the 3DS format.
Easy to integrate ARToolKit	Due to the very complex architecture of OpenSG it was not easy to integrate ARToolKit. But we have proved that it is possible.	Due to the easy extension of Open SceneGraph classes, the easy to understand architecture and the OpenGL likely display method it was easy to integrate ARToolKit into Open SceneGraph.

The feasibility study showed that both scene graph libraries can be integrated wit ARToolkit. But the weaknesses of OpenSG mentioned earlier and shown in the above table, and the intuitive and easy extendable architecture of OpenSceneGraph led to the conclusion that OpenSceneGraph is the better

choice for the AMIRE framework. Especially the complicated way to extend OpenSG fostered this decision.

13. C++ programming guidelines

13.1. Language independent rules

13.1.1. Unary operators

Correct:	`v++`
	`--v`
	`-10`

Wrong:	`-- v` `- 10`

13.1.2. Binary operator

Use always exactly one separating blank on the left and on the right side of a binary operator.

Correct:	`a + b`
	`a - b`
	`a * b`
	`a / b`
	`a & b`
	`a | b`
	`a ^ b`
	`a << b`
	`a >> b`
	`a <= b`
	`a >= b`
	`a != b`
	`a == b`
	`a && b`
	`a || b`
	`a = b * c`
	`a *= b`

Wrong:	`a  + b`
	`a- b`
	`a ==b`
	`a*b`
	`a = b*c`
	`a *=b`

13.1.3. Brackets

There is no separating blank allowed between the outer brackets and the inner expression.

Correct:	`(a + b)` `((a - b) * (a + b))`
Wrong:	`( a - b)` `( (a - b) * (a + b))`

13.1.4. Statements and statement delimiters

- Only one statement per line is allowed.
- There is no blank between the statement and its statement delimiter.

Correct:	`a++;`
Wrong:	`a++ ;` `a++; b++;`

13.1.5. Indentation

- Tabulator characters are not allowed for indentation.
- Two space characters represent one indentation level.
- The indentation level is initialised with zero.
- Entering a block (all statements between the '{' and '}' brackets) increases this indentation level by one.

13.1.6. Font

It is recommended to use a font with a fixed width like *Courier*. The C++ guidelines are optimised for such fonts.

13.1.7. Line length

- 100 characters is the maximum length of a line.
- Each single statement that is longer than 100 characters must be splitted into several lines. The indentation level of the first line is increased by two for all other lines.

13.2. Control structures

Do not use the following statements:

- Continue
- Break (only allowed for the termination of a switch case)
- Goto

The correct formatting of the control structures is described by EBNF syntax in the following sections. Additionally two important breaking rules must be defined.

- Statements line must be broken at the start of a new parameter, if the line is too long. If this rule fails the breaking rule of expressions must be applied. When the expression rule also has failed the break should be made at a place, which looks good for the programmer. The indentation level of each new line is increased by two. This means four additional indentation characters at the position of the first statement character.

- Expressions that are too long for the line must be broken at the first character of an operand expression. Higher order operand terms are preferred. The indentation position is the same position as the first character of the broken expression.

EBNF syntax of an expression:

```
EXPRESSION =
     '(' EXPRESSION ') ' OPERATOR ' (' EXPRESSION  ')'
     { OPERATOR ' (' EXPRESSION  ')' } |
     TERM { ' ' OPERATOR ' ' TERM }.
```

Example that has to be broken:	`(a && b) || (c && d)`

Correct:	`preferred break:`
	`(a && b) ||`
	`(c && d)`

	`or:`
	`(a &&`
	` b) || (c && d)`

Wrong:	`(a` ` && b) || (c && d)`
	`(a && b)` `|| (c && d)`
	`(a && b) ||` ` (c && d)`
	`(a &&` `b) || (c && d)`

13.2.1. if

EBNF syntax :

```
IF_STATEMENT =
     'if (' EXPRESSION ') {' NEWLINE {
       ADD_IDENT STATEMENT_SEQUENZE NEWLINE DEC_IDENT
     '} else if (' EXPRESSION ') {' NEWLINE
       ADD_IDENT STATEMENT_SEQUENZE NEWLINE DEC_IDENT } [
     '} else {' NEWLINE
       ADD_IDENT STATEMENT_SEQUENZE NEWLINE DEC_IDENT ]
     '}' NEWLINE .
```

If the first line is too long it must be broken by the expression rule.

Correct:	`if (expression) {`
	`  statement;`
	`}`
	
	`if (expression1 &&`
	`  expression2) {`

	`  statement;`
	`}`
	
	`if (expression) {`
	`  statement1;`
	`} else {`
	`  statement2;`
	`}`
	
	`if (expression1) {`
	`  statement1;`
	`} else if (expression2) {`
	`  statement2;`
	`}`
	
	`if (expression1) {`
	`  statement1;`
	`} else if (expression2) {`
	`  statement2;`
	`} else {`
	`  statement3;`
	`}`

Wrong:	`if (expression)`
	`{`
	`  statement;`
	`}`
	
	`if (expression){`
	`  statement;`
	`}`
	
	`if(expression) {`
	`  statement;`
	`}`
	
	`if (expression) {`
	`  statement1;`
	`}else {`
	`  statement2;`
	`}`
	
	`if (expression) {`
	`  statement1;`
	`} else{`
	`  statement2;`
	`}`
	`if (expression) statement;`

13.2.2. while

EBNF syntax :

```
WHILE_STATEMENT =
     'while (' EXPRESSION ') {' NEWLINE
       ADD_IDENT STATEMENT_SEQUENZE NEWLINE DEC_IDENT
     '}' NEWLINE .
```

If the first line is too long it must be broken by the expression rule.

Correct:	`while (expression) {`
	`  statement;`
	`}`
	
	`while (expression1 &&`
	`  expression2) {`
	`  statement;`
	`}`

Wrong:	`while (expression)`
	`{`
	`  statement;`
	`}`
	
	`while (expression) statement;`

13.2.3. for

EBNF syntax :

```
FOR_STATEMENT =
     'for (' [ STATEMENT ] ';' [ ' ' EXPRESSION ] ';'
             [ ' ' STATEMENT ] ') {' NEWLINE
       ADD_IDENT STATEMENT_SEQUENZE NEWLINE DEC_IDENT
     '}' NEWLINE .
```

If the first line is too long you must break the line after each `';'`. If one of the new lines is still too long you must break the affected line by applying the breaking rules of statements and expressions.

Correct:	`for (int i = 0; i < 10; i++) {`
	`  statement;`
	`}`
	
	`for (int realLongCounter = 0;`
	`  realLongCounter < 10;`
	`  realLongCounter ++) {`
	`  statement;`
	`}`
	`for (; i < 10; i++) {`
	`  statement;`
	`}`
	
	`for (; i < 10;) {`
	`  statement;`
	`}`

	`for (;;) {`
	`  if (expression) {`
	`   return 10;`
	`  }`
	`  statement;`
	`}`

Wrong:	`for (int i = 0; i < 10; i++)`
	`{`
	`  statement;`
	`}`
	
	`for (int i = 0;i < 10;i++) {`
	`  statement;`
	`}`
	
	`for (int i = 0;i < 10;i++)`
	`  statement;`

13.2.4. do while

EBNF-Syntax :

```
DO_WHILE_STATEMENT =
     'do {' NEWLINE
       INC_IDENT STATEMENT_SEQUENZE NEWLINE DEC_IDENT
     '} while (' EXPRESSION ');' NEWLINE .
```

If the last line is too long it must be broken by the expression rule.

Correct:	`do {`
	`  statement;`
	`} while (expression);`
	
	`do {`
	`  statement;`
	`} while (expression1 &&`
	`  expression2);`

Wrong:	`do`
	`{`
	`  statement;`
	`} while (expression);`
	
	`do statement;`
	`while (expression);`

13.2.5. switch

EBNF-Syntax :

```
SWITCH_STATEMENT =
     'switch (' EXPRESSION ') {' NEWLINE {
       ADD_IDENT 'case ' CONST ':' NEWLINE
```

```
        ADD_IDENT STATEMENT_SEQUENZE NEWLINE
        [ 'break;' NEWLINE ] DEC_IDENT DEC_IDENT } [
      ADD_IDENT 'default ' CONST ':' NEWLINE
        ADD_IDENT STATEMENT_SEQUENZE NEWLINE
        [ 'break;' NEWLINE ] DEC_IDENT DEC_IDENT ]
    '}' NEWLINE .
```

If the first line is too long it must be broken by the expression rule.

Correct:	`switch (x) {`
	`  case 1:`
	`  statement1;`
	`  break;`
	`  default:`
	`  statement2;`
	`  break;`
	`}`
	`switch (expression1 +`
	`  expression2) {`
	`  case 1:`
	`  statement1;`
	`  break;`
	`}`

Wrong:	`switch (x) {`
	`case 10:`
	`  statement1;`
	`  break;`
	`default:`
	`  statement2;`
	`  break;`
	`}`
	
	`switch (x) {`
	`  case 10: statement1; break;`
	`  default: statement2; break;`
	`}`

13.3. Names

13.3.1. Language

All names must be in English.

13.3.2. Methods, namespaces, enumerations, variables and attributes

A name must be build of lowercase words making sense and describing the variable or attribute. To separate two words in a name you must write the first letter of the second uppercase. Summarizing all characters of the name must be lowercase or a number, except the first one and all characters separating two words.

Don't use acronyms with a small number of characters like 'dba' which could mean 'data base architecture' or 'data base access' or 'data base administrator' or 'double bold arrow' and so on.
Don't use characters in front of the name that describes meta information like type or visibility.

13.3.3. Classes and types

The name is build like in the previous section. The difference it that the first character is uppercase.

13.3.4. Defined constant values and macros

All letters of defines (*#define EXAMPLE_NAME*...) have to be uppercase and the words are separated by a **'_'** character.

EBNF syntax:
UPPERCASE_IDENTIFIER = all characters are uppercase and numbers are also allowed.
DEFINE =
UPPERCASE_IDENTIFIER { **'_'** UPPERCASE_ IDENTIFIER } .

13.4. Classes

- All fields of a class and methods are sorted by static modifier, kind (attributes, constructors, destructors) and access (public, protected and private).
- All static fields are at the top of the class.
- Inside of this two blocks the fields are sorted by the kind: attributes comes first, constructors and destructors comes next and methods comes at last.
- Constructors come before destructors.
- Virtual methods (dynamic binding at runtime) come before non-virtual methods (static binding at compile time).
- All kind of fields are sorted by their access: 'public' comes first, 'protected' next and 'private' at last.
- 'public', 'protected' and 'private' are on the same indentation level as 'class'
- Inner classes are indented by one indentation level (two additional space characters).
- A method names starts with a lowercase character.
- Get- and set-methods should be used to access attributes (get-methods for Boolean value can be named with 'is…' instead of 'get…').
- Virtual methods should only be used for methods that really need dynamic binding at runtime.
- Abstract virtual methods should not be implemented (use instead an assignment of zero to the method table) to ensure (by the compiler) that the abstract class and method are not directly used.
- The use of inline methods is allowed to optimise the performance or to create template classes.
- Class and type names must begin with an uppercase character.

EBNF syntax:

```
CLASS =
     'class ' CLASS_NAME ' : ' BASE_CLASSES ' {'
     'public:' NEWLINE STATIC_ATTRIBUTES SEPERATINGLINE
     'protected:' NEWLINE STATIC_ATTRIBUTES SEPERATINGLINE
     'private:' NEWLINE STATIC_ATTRIBUTES SEPERATINGLINE
```

```
     'public:' NEWLINE STATIC_METHODS SEPERATINGLINE
     'protected:' NEWLINE STATIC_METHODS SEPERATINGLINE
     'private:' NEWLINE STATIC_METHODS SEPERATINGLINE

     'public:' NEWLINE ATTRIBUTES SEPERATINGLINE
     'protected:' NEWLINE ATTRIBUTES SEPERATINGLINE
     'private:' NEWLINE ATTRIBUTES SEPERATINGLINE

     'public:' NEWLINE CONSTRUCTORS SEPERATINGLINE
     'protected:' NEWLINE CONSTRUCTORS SEPERATINGLINE
     'private:' NEWLINE CONSTRUCTORS SEPERATINGLINE

     'public:' NEWLINE METHODS SEPERATINGLINE
     'protected:' NEWLINE METHODS SEPERATINGLINE
     'private:' NEWLINE METHODS SEPERATINGLINE

     'public:' NEWLINE INNER_CLASSES SEPERATINGLINE
     'protected:' NEWLINE INNER_CLASSES SEPERATINGLINE
     'private:' NEWLINE INNER_CLASSES
     '};' NEWLINE .

SEPERATINGLINE = if more fields in class then NEWLINE otherwise
     nothing .

ATTRIBUTES =
     INC_INDENT
     ATTRIBUTES NEWLINE { ATTRIBUTE NEWLINE }
     DEC_INDENT .

CONSTRUCTORS =
     INC_INDENT
     CONSTRUCTOR NEWLINE { CONSTRUCTOR NEWLINE }
     [ DESTRUCTOR NEWLINE { DESTRUCTOR NEWLINE } ]
     DEC_INDENT .

METHODS =
     INC_INDENT
    'virtual ' METHOD NEWLINE { 'virtual ' METHOD NEWLINE }
    SEPERATINGLINE
     METHOD NEWLINE { METHOD NEWLINE }
     DEC_INDENT .

INNER_CLASSES =
     INC_INDENT
     INNER_CLASS SEPERATINGLINE
     { INNER_CLASS SEPERATINGLINE }
     DEC_INDENT .
```

Example:

```
class MyClassInherited : public MyClass {
  public:
    static int x;

  protected:
    static int y;

  private:
    static int z;

  public:
    static int getX();

  protected:
    static int getY();

  private:
    static int getZ();

  public:
    int a;
    int b;

  protected:
    int c;
    int d;

  private:
    int e;
    int f;

  public:
    MyClassInherited();

  protected:
    MyClassInherited(int y);

  private:
    MyClassInherited(bool z);

  public:
    virtual ~MyClassInherited();
    virtual void h(int x = 1) = 0;

    void g();

  protected:
    virtual void i() = 0;
```

```
    void j();

  private:
    void k();
    void l();
};
```

13.5. Namespaces

Use namespaces to ensure unique class and function names. The namespace doesn't affect defined macros of constants. So the name of a define is very important to prevent several defines with the same name. Instead of defines you must prefer enumerations (maybe in combination with a type definition).

Example: Instead of

```
#define FRUIT_APPLE 0
#define FRUIT_PEACH 1
#define FRUIT_PEAR 2

void drawFruit(int fruit);
```

use this

```
namespace fruits {
  typedef enum {
    apple = 0,
    peach,
    pear
  } Fruit;

  void drawFruit(Fruit fruit);
}
```

The name of namespaces must reflect the purpose of this package. It must begin with a lowercase letter. Also the project name should be included in the namespace. The best way to do this is to create a namespace with the project name containing all other namespaces (you can also add the company name to the namespace, when the project name is not unique).
In source files the use of *'use namespace NAMESPACE_NAME;'* is allowed to reduce the indentation level and to use the namespace content without the complete qualifier.

EBNF syntax:

NAMESPACE = 'namespace' *NAMESPACE_NAME* '{' *NEWLINE* *INC_INDENT* *NAMESPACE_CONTENT* *DEC_INDENT* *NEWLINE* '}' *NEWLINE* .

NAMESPACE_NAME = name describing the functionality provided by the namespace content (first word is completely lowercase the next words are lowercase except each first character is uppercase) .

NAMESPACE_CONTENT = contains other namespaces, classes, enumerations, types, functions, methods, and so on .

13.6. Header and Source files

To ensure that a header file can be included more than once you must provide information to the pre-processor that this file is already included. Additionally in header file only includes with '<' and '>' brackets are allowed to ensure that the compiler is able to find the header files. In source file you must use includes with '"' characters for project internal headers to speedup the time of compilation. For including header files of other projects or libraries you must use the includes with '<' and '<' brackets.

13.6.1. Prevent re-including

The prevention of re-including is based on the pre-processor. It uses defines and unique names for the header files. So the first problem is to get a unique name of the header file. Therefore use only one namespaces path in a header file. Based on this convention you can use the namespace and header file name to create a unique define name for the header file.

EBNF syntax:

```
UNIQUE_HEADER_FILE_NAME =
     '_' { UPPERCASE_NAMESPACE_NAME '_' }
     UPPERCASE_HEADER_FILE_NAME .
```

At the top of the header file you must ask the pre-processor if the unique header file name is not defined (*#ifndef*). The pre-processor will only consider the next lines when it isn't defined. So we have to define the unique header file name. After the definition you can put the content of the header file and at the end you must end the pre-processor if-command (*#ifndef*) with *#endif*.

Example:

```
#ifndef _NAMESPACE_FILENAME_H
#define _NAMESPACE_FILENAME_H

// content ...

#endif
```

13.6.2. Head information

The header and source file must contain a head with the information about:

- Author
- Creation date
- Modification date
- Small description
- Copyright

- Licence
- …

Example of a head information:

```
/* Description:   This is a sample demonstrating the head
 *                information of a header or source file.
 *
 * Copyright by:  FH Hagenberg
 *                Studiengang Medientechnik und -design
 *                Hauptstrasse 117
 *                A-4232 Hagenberg
 *                mailto://mtd@fh-hagenberg.at
 *                http://mtd.fh-hagenberg.at
 *
 * Licence:       See Licence.txt
 *
 * Author:        Juergen Zauner
 *                mailto://juergen.zauner@fh-hagenberg.at
 *
 * Co-authors:    Werner Hartmann
 *                mailto://whartmann@faw.uni-linz.ac.at
 *
 * Verision:      0.0.1
 *
 * Created:       2002-06-18
 * Last modified: 2002-06-21
 */
```

13.7. Documentation

Documentation is always a trade-off between too much information and too less information. In the source code for example only a required minimum should be documented. When the documentation for one specific problem gets too long it often points to a problem concern the source code quality. Try to think over the affected code and maybe you find another solution that is more satisfying and requires less documentation. The best code is the code that explains itself and doesn't need any comment.

The documentation of header files is more important than the documentation of source files. Source files are only visible and maintained by a small set of developers. So for the maintenance process it's more important to get a clean and high quality source code understandable for each developer. Header files are used by developers who want to use the functionality of the library. Therefore the functionality and behaviour of the header file content (classes, methods, attributes, variables, types, macros, and so on) must be explained by the documentation. The behaviour also includes the behaviour at a failure situation and not only the default behaviour.

An important problem of documentation is its validity. Documenting virtual method in each derived class would create a huge set of redundant documentation, which has to be maintained when the default behaviour of the base class changes. The solution for this problem is to document only the virtual methods of the base classes. Additional behaviour that differs from the behaviour of the base class method must be documented at the method of the derived class.

The use of documentation tools that create HTML documents out of the source and header files should be used. For example:

- doxygen (http://www.stack.nl/~dimitri/doxygen/index.html),
- ccdoc (http://www.joelinoff.com/ccdoc/) or
- doc++ (http://www.zib.de/Visual/software/doc++/index.html).

When you have decided to use one tool it should be used for the whole project to ensure an unique look and feel. They often support to copy the documentation of base class methods to undocumented methods of the derived classes. You must add a kind of reference for the undocumented method of a derived class pointing to the base documentation if the tool does not provide the copy functionality.

Appendix A –References

[TheAno01] **Theauthor, A.; Anotherauthor, B.:** *Title of Publication*, Foobar Press Publications, New York, 2001

[BlaBil99] **Blanding B.; Billinghurst M.; Kato H.; May R.:** *ARToolKit*, Technical Report, Hiroshima City University, December, 1999

[deIp02] **L. de Ipina:** *Visual Sensing and Middleware Support for Sentient Computing*, University of Cambridge, Downing College, http://www-lce.eng.cam.ac.uk/~dl231/PhDThesis/thesis.pdf, pp. 42-96, 2002

[GamHel95] **Gamma, E.; Helm, R.; Johnson, R; Vlissides, J.:** *Design Patterns in Smalltalk MVC, in Design Patterns*, Addison Wesly Longman Inc, pp. 4-6, 1995

[Hamilt02] **Hamilton, G.:**, *JavaBeans*, Sun Microsystems Inc http://java.sun.com/products/javabeans/docs/spec.html, 2002

[Strous00] **Stroustrup, B.:** *Run-Time Type Information, in The C++ Programming Language (Third Edition)*, Addison Wesly Longman Inc, pp. 407-418, 2000

[Green02] **Green, D.:** *The Reflection API*, Sun Microsystems, http://java.sun.com/docs/books/tutorial/reflect/index.html, 2002

[OpenSceneG02] **OpenSceneGraph:** www.openscenegraph.org, 2002

[OpenSG02] **OpenSG:** www.opensg.org, 2002

[Perf02] **Performer:** http://www.sgi.com/developers/devtools/apis/performer.html, 2002

[Trollt02] **Trolltech Inc:** *Qt 3.0 Whitepaper*, Trolltech Inc, http://www.trolltech.com/products/qt/whitepaper/whitepaper.html, 2002

[SunMic02a] **Sun Microsystems Inc:** BeanBox, Sun Microsystems Inc, http://java.sun.com/docs/books/tutorial/javabeans/beanbox/index.html, 2002

[SunMic02b] **Sun Microsystems Inc:** *Writing Event Listeners*, Sun Microsystems Inc, http://java.sun.com/docs/books/tutorial/uiswing/events/index.html, 2002

[Rabin00] **Rabin, S.:** *Designing a General Robust AI Engine, In Game Programming Gems,* Charles River Media, pp. 221-236, 2000.

Appendix B – Glossary

Base gem: is a gem allowing the object-oriented abstraction of a gem groups functionality.

Basic property type: is a basic type like integer in C++. The framework provides the following basic property type: boolean property type, character property type, string property type, integer property type, float property type and double property type.

Complex gem: is a gem network packaged into a single gem.

Component: is a cloneable, configurable and connectable prototype providing functionality like a gem. It is build with gems and has generic interfaces, which are common for all components.

Component instance: is a cloned instance of the corresponding component.

Component network: is a set of component instances connected among themselves. It provides a functionality resulting out of the connection between the component instances. Components are connected by using in- and out-slots.

Composed component: is a component network packaged into a single gem.

Gem: provide the functionality required by the developer for the implementation of the application. The definition of the gem could exist as snippets of source code part of a publication or another application or as a theoretical work in a publication. An implementation of the gem must be integrated into the framework.

Gem category: is a set of gem groups providing the same behavior. Convention of the gem look and feel of the gems in the category must be provided by the framework. This includes naming conventions for methods and the behavior in error cases. For example a loader gem category will contain an image loader gem group, scene node loader gem group and other loader gem groups.

Gem group: is a set of gems with the same basic functionality. The framework must provide an abstraction mechanism like a base-gem or plug-in gem. For example an image loader gem would be part of the image loader gem group.

Gem network: is a set of gem instances connected among themselves. It provides a functionality resulting out of the connection between the single gem instances.

Framework: holds everything together. It provides basic functionality and conventions for gems and components. It integrates all gems and components provided by the AMIRE project.

High-level gem: is a gem available as source code and could be integrated into the framework. This can be source snippets or complex gems, which are build out of other gems. The gem or framework developers will maintain the high-level gems.

In-slot: is an interface of a component receiving a property of a specific type. It is accessible by a name.

Low-level gem: is a main library gem provided by other frameworks or libraries. For example a sound library as OpenAL provides low-level gems for playing 3D sound sources. Basically it is not possible to change or manipulate the sources of low-level gems. This means the maintenance work has to be done by the provider of the library.

Main library gem: is a gem that must be integrated into the MR framework. Due to its dependencies on the framework it can't be provided as a separated gem outside of the framework.

Out-slot: is an interface of a component emitting a property of a specific type. It is accessible by a name.

Plug-in gem: is part of a mechanism provided by the framework to abstract the functionality of a gem group. It allows changing the implementation of the functionality easily by changing the gem, which provides the functionality.

Property: is the generic representation of data provided by the framework. Therefore property types are required to interpret the data correct.

Property type: is the meta information of the property. It describes the internal structure and the semantic of the data contained in the property. The framework provides basic, structured and vector property types.

Vector property type: is like the vector template in C++. A property of this type contains an array of properties with the same property type. Additionally a vector property can grow.

Structured property type: describes a structured data like a class in C++. The field properties are accessible by their field names.

State listener: will be notified by the state machine when a trigger state of the state listener occurs.

State machine: is a model with a state visible to other objects. It handles a set of state listeners. Each of them is registered for a specific trigger state. The state listener is notified, when the state machines current state is set to its trigger state by an external or internal state change.

Trigger state: is a state machines special state, which will initiate a notification of a state listener.