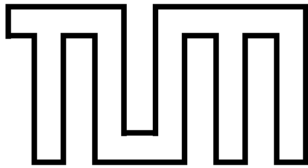
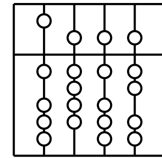


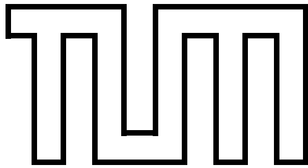
Technische Universität
München
Fakultät für Informatik



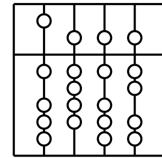
Systementwicklungsprojekt

Development of a 3D-Viewer Component for DWARF

Otmar Hilliges



Technische Universität
München
Fakultät für Informatik



Systementwicklungsprojekt

Development of a 3D-Viewer Component for DWARF

Otmar Hilliges

Aufgabensteller: Univ-Prof. Gudrun Klinker, Ph.D.

Betreuer: Dipl.-Inf. Christian Sandor

Abgabedatum: 21. April 2004

Erklärung

Ich versichere, dass ich diese Ausarbeitung des Systementwicklungsprojektes selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 21.April.2004 Otmar Hilliges

Abstract

This document describes a 3D-viewer component for DWARF[9]. In the course of the ARCHIE[1] project a new viewer had to be developed, because previous systems developed at the chair for applied software engineering[2] have shown several problems in available solutions.
Among them are the lack of a cross-platform solution, performance problems, different interfaces for each browser and missing features like real stereo modes and video see-through capability.
The goal of my work was the development of a lightweight 3D-viewer as a DWARF component and the support of different 3D-scene description standards. Also different stereo modes and a video see-through mode have been developed.

[1]A group effort of 7 diploma and undergraduate students at Technische Universität München
[2]Technische Unviversität München, Lehrstuhl für angewandte Software Technik

Contents

1 Introduction 5

1.1 Motivation . . . 5

1.2 Goals . . . 6

1.3 Structure of this Document . . . 6

2 Thesis Context 7

2.1 DWARF . . . 7

2.2 ARCHIE . . . 8

2.2.1 ARCHIE Problem Statement . . . 8

3 Requirements Analysis 11

3.1 Scenarios . . . 11

3.1.1 ARCHIE Scenarios . . . 11

3.2 Use Cases . . . 14

3.3 Requirements . . . 16

3.3.1 Functional Requirements . . . 16

3.3.2 Nonfunctional Requirements . . . 18

3.3.3 Pseudo Requirements . . . 18

4 Related Work 19

4.1 Scenegraphs . . . 19

4.2 Scenegraph-based Frameworks . . . 19

4.3 Open Inventor . . . 21

4.3.1 Nodes . . . 22

4.3.2 Interaction Management . . . 22

4.4 Studierstube . . . 24

5 Implementation **25**
5.1 Static Model . . . 27
5.1.1 Communication and Event processing . . . 28
5.1.2 Scenegraph Management . . . 31
5.1.3 Rendering . . . 32
5.2 Dynamic Models . . . 35
5.2.1 Object Model . . . 36
5.2.2 Sequence Diagrams . . . 38
5.3 Benchmarks . . . 39

6 Conclusion **42**
6.1 Results . . . 42
6.2 Lessons Learned . . . 43
6.3 Future Work . . . 43
6.3.1 Technical Improvements . . . 43
6.3.2 Functional Improvements . . . 44
6.3.3 Further Leading Ideas . . . 44

Bibliography **45**

List of Figures

3.1 HMD calibration with a pointing device 12
3.2 Modeling and Form Finding 13
3.3 Presentation of a planned building to an audience 14
3.4 Line interleaved stereo 17
3.5 Video see through while second user is modeling 17

5.1 The DWARF user interface architecture 26
5.2 The interface of the viewer displayed in DWARF *needs & abilities* 27
5.3 The class structure of the 3D-viewer. 28
5.4 The simplified 3D-viewer architecture. 29
5.5 The *ViewerFacade* class, its attributes and operations. 30
5.6 The structured event push consumers. 30
5.7 The*SceneGraphManagement* package. 31
5.8 The*Rendering* package. 32
5.9 The initial scene graph stored in *SoDwarfViewer*. 32
5.10 The *SoDwarfViewer* class. 33
5.11 The Stereo modes available. 35
5.12 The service description for the 3D-Viewer in XML 36
5.13 Object model resulting from service description given in 5.12. 37
5.14 Setting the virtual viewpoint. 38
5.15 Adding a 3D object to the scene graph. 39
5.16 Replacing the whole scene graph. 39
5.17 Frames per second according to the nr. sheep displayed in the viewer (2Hz update rate each sheep). 40

5.18 Frames per second according to the nr. sheep displayed in the viewer (10Hz update rate each sheep). 41

Chapter 1

Introduction

Modern life wouldn't be imaginable without the computers that help us to manage and process all the information and data needed to solve our problems. Still most computer software is an abstraction of reality due to the limitations of todays computers.
Augmented reality (AR) is a possibility to reduce the level of abstraction and therefore reduce the amount of redundant work.
This is achieved, among others, by new means of human computer interaction (HCI). Users no longer have to sit in front of desktop fixed computers and work with a mouse and a keyboard, but instead can move free in space and interact directly with the virtual world as they are used to interact with the real world. As outcome users don't have to concentrate on computers, software and the usage of them any longer. Instead users get the information they need superimposed on the real world and can use tangible objects to manipulate those informations.
AR applications demand a broad variety of new technologies such as tracking devices, network technology and new input and output devices. In this thesis I concentrate on the output part of such applications, especially on the display of three dimensional graphics.

1.1 Motivation

This 'Systementwicklungs Projekt' is part of the DWARF project which is developed at the chair for applied software engineering of TUM since the year 2000. DWARF[9] is an architecture for **D**istributed **W**earable **A**ugmented **R**eality **F**ramework based applications. It supports rapid prototyping of AR applications and was presented at the ISMAR'01. Since then several prototypes have been developed such as TRAMP, FixIT and Sheep[21].

All projects suffered under the poor performance of the graphical output system.

Mainly because off the problems described above. This 'Systementwicklungs Projekt' is also part of the next DWARF project: ARCHIE an architectural design system which supports a 3D modeling environment. To make 3D modeling in real-time possible, a sophisticated stereo 3D output component, which should be the outcome of this project, is very important.

1.2 Goals

With my work I want to address several problems in Augmented reality user interfaces concerning the graphical presentation subsystem. Here I just want to give a short overview which aspects have been covered.

1. technical:
   - lightweight Augmented reality browser
   - render three dimensional scenes in various file formats (VRML/Inventor)
   - different Stereo/Mono modes for different presentation purposes
   - video see-through
2. capabilities:
   - addition and removal of objects to and from the scene
   - control position and orientation of objects in the scene
   - overlay scene with two dimensional information (e.g. text)
   - calibrate the browser so that the scene is rendered correctly

1.3 Structure of this Document

In this document I will show why a new 3D-viewer component was needed for DWARF and which requirements have to be fulfilled by such a component (Chapter 1 through Chapter 3).
In Chapter 4 I will inspect and explain related work and familiar frameworks.
In Chapter 5 I will show how the requirements and ideas gathered in the previous chapters have been implemented. In Chapter 5 I will also describe how the 3D-viewer component interacts with the surrounding DWARF environment and how to use it in Augmented reality systems.

Chapter 2

Thesis Context

To get a deeper understanding of the requirements for the component we have to take a look at the target environment(DWARF) for the component and at the context within it is developed(ARCHIE).

2.1 DWARF

The 3D-viewer is designed as component of the DWARF [9][11] framework which is developed at the Technische Universität München since the year 2000. The design of the Framework is geared towards distributed, ubiquitous Augmented reality computing. DWARF consists basically of the distributed service manager which locates and connects several services (e.g. input devices, 3D-viewer or tracking devices) dynamically.

The combination of distributed and ubiquitous computing concepts in DWARF allows developers to create new Augmented reality systems within short time because they can reuse already existent components and combine them in new ways or add new components to create new functionality. The components in DWARF are named *services*

The connections between the services are handled via *needs* and respectively *abilities*. Whenever an ability is offered by one service which matches a need by another service those two are connected and can communicate amongst each other.

2.2 ARCHIE

As stated in 1.1 DWARF was already used to build various Systems among them TRAMP[1], FixIt[2] and Sheep[3][21].

This section[4] introduces the ARCHIE project. The name is an acronym for **A**ugmented **R**eality **C**ollaborative **H**ome **I**mprovement **E**nvironment.

ARCHIE is an interdisciplinary project between the Universität Stuttgart represented by Manja Kurzak, a student graduating in architecture [17] and a team consisting of seven members from the Technische Universität München. Manja Kurzak provided information about design processes and the planning of the construction of e.g. public or private buildings. Her information is summarized in the following problem statement section.

This project provided a starting point for requirements elicitation on the different single projects of all team members. It was intended to use ARCHIE in a prototypical implementation to give a proof of the requested components and their underlying concepts. To build an application with full functionality for architectural requirements was never a project goal.

The realized concepts have been shown to stakeholders like real-world architects in a live demonstration of multiple scenarios. In new areas like Augmented reality new requirements are often generated by the client when the capabilities of the technology get apparent. To prove the flexibility and extendibility of the new DWARF components the chosen scenarios are partly independent.

2.2.1 ARCHIE Problem Statement

A *Problem statement* is a brief description of the problem the resulting system should address[12].

Usually a number of people with different interests are involved in the development process of a building.
The potential buyer has to mandate an architectural office to initiate the building

[1] Traveling Repair and Maintenance Platform, undergraduate student project, fall semester 2001
[2] undergraduate student project, summer semester 2002
[3] Demo at ISMAR 2002 Darmstadt, Germany
[4] taken from the common chapter in [26][16][22]

process because the process is too complex to handle for himself. A mediator is responsible to represent the interest of the later building owners towards the architect. The architect assigns work to specialized persons such as for example, technical engineers for designing plans of the wiring system.
Although the later building owner is the contact person for the architects office, he is only one of the many stakeholders interested in the building. Furthermore landscape architects have to approve the geographic placement of the new building in the landscape. Last but not least a building company has to be involved as well.

The architectural process itself is divided into multiple activities which are usually handled in an incremental and iterative way, as some specialized work goes to extra technical engineers, who propose solutions, but again have to reflect with the architects office.
After taking steps of finding and approximating the outer form of the building fitting in it's later environment, the rudimentary form is enhanced to a concrete model by adding inner walls, stairs and minor parts. This is followed by adding supportive elements to the model like water- and energy-connection, air-conditions, etc. As mentioned, this work always has to be reflected to the architect, because problems might occur during the integration of the different building components. For example the layout of pipes might interfere with the layout of lighting fixtures or other wiring. Spatial representation enhances pointing out problematic situations.
When the plans are nearly finished, the builder needs the possibility to evaluate the plan feasibility and if in the end all issues are resolved, all necessary plans can be generated and the builder can start his work.
In addition to that the building owner always needs view access to the model during the design phase. He wants to see his building in its environment. End users should be given the option to give feedback during the design phase too. So, the architects office receives feedback from many participants about their plans.

There are some entry points for Augmented reality. The ARCHIE project delivers proof of concept for these aspects.
The benefits of the old style architectural design with paper, scissors and glue allows direct spatial impressions, while modern computer modeling does not provide these feature. Augmented reality can bring these back to computer modeling.
Since preliminary, cardboard box models can not get scaled to real size and virtual models reside on a fix screen, public evaluations are difficult to handle. Via abstraction of position and orientation independent sliders could be used to modify views. But 3D steering of virtual viewpoints is not as intuitive as just turning a viewers head. Adding tangible objects as cameras provide familiar access to evaluation features.
User interactions with modern architectural tools require practice. So intuitive input devices would be useful.
Also in place inspection of building plans and models would be a great benefit for all

participating persons.

Chapter 3

Requirements Analysis

Requirements Elicitation focuses on describing the purpose of the system. The client, the developers, and the users identify a problem area and define a system that addresses the problem[12].

In this chapter I will describe the scenarios that evolve from the problem statement (2.2.1 and, out of that, the requirements which the 3D-viewer component has to fulfill

3.1 Scenarios

A scenario is a concrete, focused, informal description of a single feature of the system from the viewpoint of a single user[12].

3.1.1 ARCHIE Scenarios

This section describes the Augmented reality relevant scenarios of the ARCHIE system 2.2. The following list does not describe a full architectural system, because the ARCHIE project is only intended to be a baseline for the development of reconfigurable DWARF services[1].

Scenario: **Calibrating the Devices**

Actor instances: `Bridget:User`

Flow of Events: 1. When she starts the calibration method with her iPaq she also needs to have the 3DOF pointing device in her hand.

[1]Note that this are only parts of the ARCHIE Scenario which are relevant to the viewing component

2. Bridget can now see the current virtual 3D scene not calibrated on the 2D image plane. In addition to that the calibration scene appears superimposed in her HMD, too. And she is asked to align the peak of the 3D pointing device with the corresponding 2D image calibration point. Once Bridget aligned the points properly, she confirms the measurement by touching her touch pad glove.
3. As the calibration method needs at last six measuring points to calculate the desired projection parameters, Bridget will be asked to repeat the last step for several times.
4. After confirming the last calibration measurement the newly calculated calibration parameters will be transmitted to the viewing component.
5. Now her HMD is newly calibrated and can augment her reality. So the tracked real objects can be overlayed by corresponding virtual objects in front of her.

Figure 3.1: HMD calibration with a pointing device

Scenario: **Modeling and Form Finding**

Actor instances: `Charlotte, Alice:User`

Flow of Events:
1. Alice and Charlotte start the ARCHIE modeling application and their HMD viewing services.
2. As the system is initialized, both see the environment of the later building site.
3. Alice takes a tangible object, moves it besides another already existing building and creates a new virtual wall object by pressing the

create button on her input device.

4. Charlotte takes the tangible object, moves it to the virtual walls position and picks up the virtual wall by pressing the select button on her input device.
5. Charlotte chooses the new position of the wall and releases it from the tangible object.
6. Both continue their work and build an approximate outer shape of a new building.

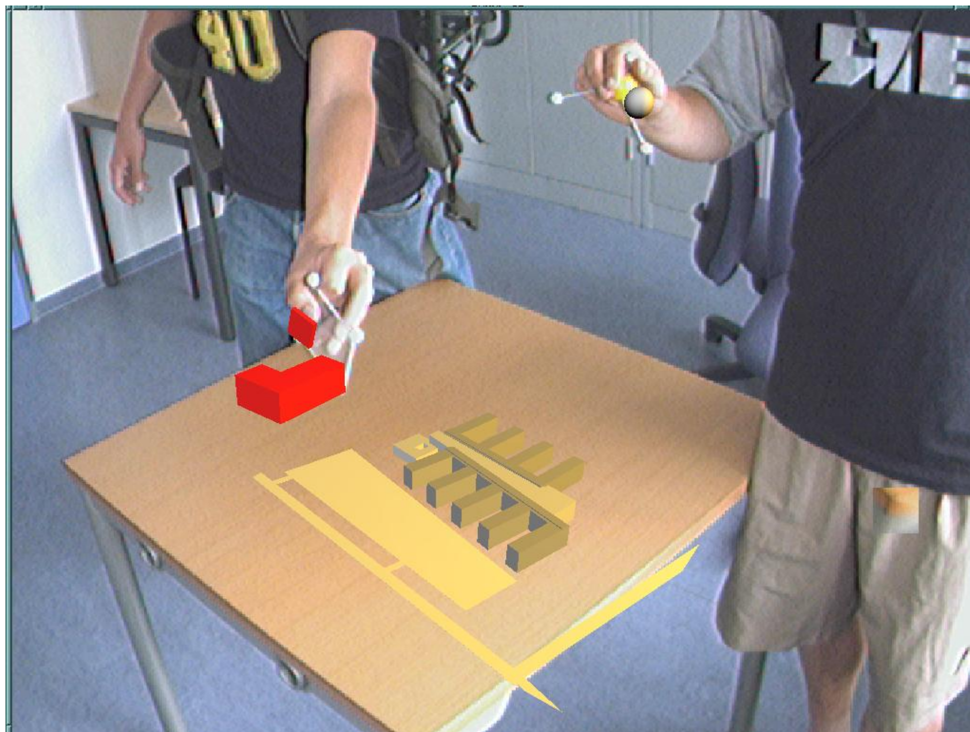


Figure 3.2: Modeling and Form Finding

Scenario: **Presentation**

Actor instances: `Alice:User`

Flow of Events:

1. Alice starts the system, but instead of the previous used HMD, now a video beamer view is started, providing scenes as seen from a tangible camera object.
2. Alice takes this camera and moves it around the virtual model of the planned building.
3. The public can get a spatial understanding of the proposed building which is displayed on the beamer screen. The model shown in

figure 3.3 is rendered in anaglyphic red-cyan 3D. For a realistic 3D view the visitors need to wear the corresponding red-cyan glasses.

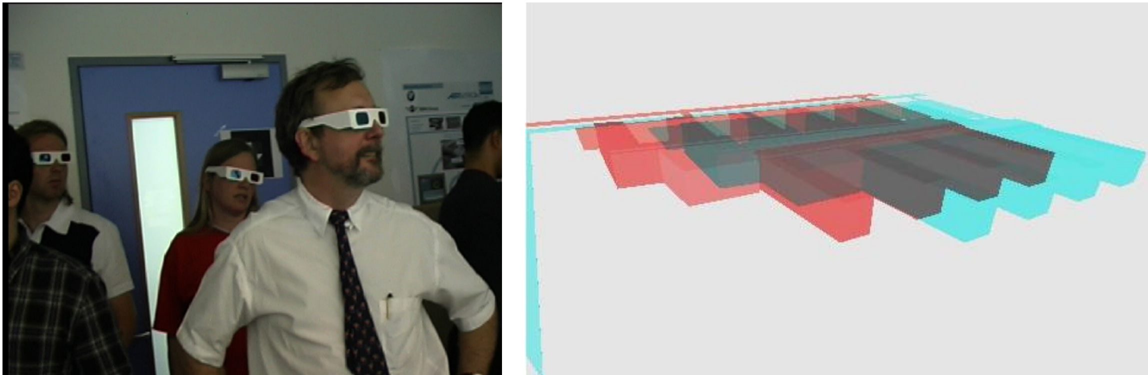


Figure 3.3: Presentation of a planned building to an audience

3.2 Use Cases

Use cases are used during requirements elicitation and analysis to represent the functionality of the system. Use cases focus on the behavior of the system from an external point of view.[12]

The Scenario described in 3.1.1 leads to following use cases which describe the functionality of the 3D-viewer.

Use Case: **Start the viewer component**

Initiated by: `User`

Communicates with:

Flow of Events:

1. *(Entry condition)* None.
2. The `User` starts the viewer either providing a initial scene (e.g. the building site loaded from a local file) or waits until the scene is provided by the *model service. further the `user` can choose between the different stereo modes and switch video see-through on or off.*
3. *(Exit condition)* The viewer is up and running showing the initial scene using the desired render mode.

Use Case: **Replace scene**
Initiated by: `User`
Communicates with:
Flow of Events:
1. *(Entry condition)* Viewer is up and running.
2. The `user` gives a command to replace the complete scene using the `input subsystem` (e.g. speech input, button).
3. *(Exit condition)* The complete scene is replaced with the provided scene.

Use Case: **Superimpose scene**
Initiated by: `System`
Communicates with:
Flow of Events:
1. *(Entry condition)* Viewer is up and running.
2. The `system` gives a command (sends an event) to overlay the current 3D scene with two dimensional information.
3. *(Exit condition)* The (dynamic) 3D scene is overlayed with 2D information which is display fixed (stays in the same position).

Use Case: **Add virtual object**
Initiated by: `User`
Communicates with:
Flow of Events:
1. *(Entry condition)* Viewer is up and running.
2. The `user` gives a command to add a virtual object to the scene using the `input subsystem` (e.g. speech input, button, tangible device).
3. *(Exit condition)* A new virtual object appears in the scene.

Use Case: **Remove virtual object**
Initiated by: `User`
Communicates with:
Flow of Events:
1. *(Entry condition)* Viewer is up and running. Further the scene contains objects that have been created in advance.
2. The `user` gives a command to remove a virtual object from the scene using the `input subsystem` (e.g. speech input, button, tangible device).
3. *(Exit condition)* A virtual object disappears from the scene.

Use Case: **Select virtual object**
Initiated by: `User`
Communicates with:
Flow of Events:

1. *(Entry condition)* Viewer is up and running. Further the scene contains objects that have been created in advance.
2. The `user` gives a command to select a virtual object from the scene using the `input subsystem` (e.g. speech input, button). further the `user` has got a tracked, tangible device with which the virtual object shall be controlled.
3. *(Exit condition)* The position and rotation of the virtual object are now adapted to the values of the tracked, tangible object.

Use Case: **Deselect virtual object**
Initiated by: `User`
Communicates with:
Flow of Events:

1. *(Entry condition)* Viewer is up and running. Further the scene contains objects that have been created in advance and at least one that is controlled by an real, tracked object.
2. The `user` gives a command to deselect a virtual object from the scene using the `input subsystem` (e.g. speech input, button).
3. *(Exit condition)* The position and rotation of the virtual object are not longer adapted to the values of the tracked tangible object (the object stays in it's position).

3.3 Requirements

3.3.1 Functional Requirements

Functional requirements describe the interactions between the system and its environment independent from its implementation[12].

File import. The viewer must be able to read in 3D scenes from files in various formats of 3D description standards such as VRML97[10], VRML2.0[15], Open Inventor[24]. Those scenes are used as background (e.g. building site)

Anaglyphic stereo. The viewer must provide a anaglyphic stereo mode. The scene has to be rendered twice once in red the second time in cyan with an appropriate offset so that it appears in 'real' 3D stereo to the spectator (see figure 3.3).

Line interleaved stereo. Also a line interleaved stereo mode must be implemented so that color preserving stereo with special hardware (e.g. virtual i/o's i-glasses HMD [8]) is possible. Line interleaved stereo provides a better stereo sensation in term of depth impression (see figure 3.4). To produce a line interleaved signal the scene has to be rendered again twice. In the first pass all lines with even linenumbers will be stamped out, in the second pass all lines with odd linenumbers will be stamped out. The signal is then put together again by special hardware to an stereo 3D image.

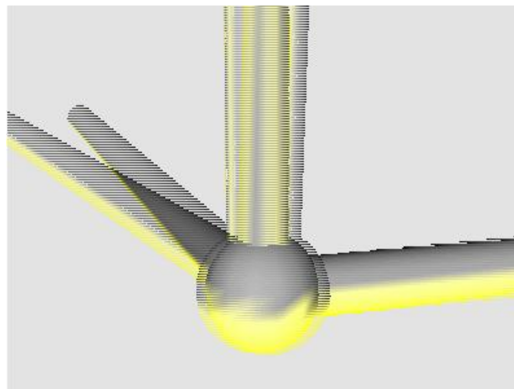


Figure 3.4: Line interleaved stereo

Video see through. For presentation purposes the viewer must provide the possibility to render a life video stream (from a firewire camera) in the background and overlay it with the 3D scene (see figure 3.5).

Figure 3.5: Video see through while second user is modeling

Add/remove objects. When it comes to means of interactivity the system must provide a way to add objects (e.g. walls, sheep) to the scene and remove them if not longer needed and or wanted.

Control objects. Again this is needed to allow interaction with the scene. To control objects means that the user and/or the system (e.g. *tracking subsystem* or autonomous services like *sheep service*) can modify their position and orientation also changing properties like size or color must be possible.

Calibration. For a correct stereo and impression and for a good matching of virtual and real world it must be possible to calibrate the (virtual) camera accordingly to the outcome of the external Calibration algorithm described in [25].

3.3.2 Nonfunctional Requirements

Nonfunctional requirements describe user-visible aspects of the system that are not directly related with the functional behavior of the system[12].

Fast Rendering. To ensure a smooth graphical impression and usability of the system the 3D-viewer component must render the complete scene with 25 fps.

Lightweight. Since DWARF is strongly focused on modularity the viewer itself shouldn't contain more then the absolutely necessary functionality. This means that its only possible to tap the full potential of the viewer when it's used in a DWARF context.

3.3.3 Pseudo Requirements

Pseudo requirements are requirements imposed by the client that restrict the implementation of the system[12].

DWARF. DWARF has to be used as base framework.

Chapter 4

Related Work

In this chapter I will take a look on related work in means of Object-Oriented 3D computer graphics and Augmented reality view components and which of our requirements are fulfilled by those.

4.1 Scenegraphs

The Scenegraph is the core concept of describing three dimensional scenes for Augmented reality systems. Scenegraphs are (hierarchic) graphs containing nodes of various types, including geometric primitives (cones, spheres, boxes). Those primitives can be grouped together to build more complex shapes. The Scenegraph also contains light sources, views and behaviors so it can store the whole scene and all the information that is needed to render it.

4.2 Scenegraph-based Frameworks

In this section I want to compare several graphical frameworks that all relay on the Scenegraph context.

Java3D[4] is a a optional package in addition to the Standard Java API. Java3D is a 3D Scenegraph based graphics programming API. The API includes Interfaces for creation of Scenegraphs (describing 3D scenes) and manipulating the Scenegraph. Further the API delivers a set of methods to do all necessary mathematical operations on vertices and matrices. The biggest advantage of Java3D is the mechanism of describing behavior. The API provides a *Behavior* class which encapsulates behavior into an reusable entity. Every objects behavior that is subclassed from the behavior class can be described with basically two methods:

- *wakeUpOn* in this method the programmer can specify the criteria upon which the *processStimulus* method is called, which describes the behaviors mechanic.
- *processStimulus* in this method the actual behavior is implemented (e.g. rotation of an object according to the value of an angle input field)

Since the behavior is always applied to a Transform Group the actual targets geometry is unimportant. This easy concept allows the programmer to model complex and powerful behavior.

VRML[10] stands for Virtual Reality Modeling Language. It uses a Scenegraph to store the scene data and provides a scripting language to model interactions. VRML is used in several AR/VR systems because of following reasons:

- Being a common ISO standard makes VRML widely used and accepted. Therefore a big community and lots of material (books, web sites and manuals) exist as source of information.
- There are a lot of browsers available to view VRML files.
- There is a broad variety of authoring tools available. That makes it easy to create 3D scenes quickly.

VRML has been used in previous DWARF systems and several problems have occured with the available browsers (for a detailed description of VRML's limitations see [20] and for a discussion of available VRML Browsers see [18]).

OpenSG [5] is another open source 3D API developed since 1999. It's main focus lies on portability, multi-threading support and extensibility. The central part and its biggest advantage is the possibility to manipulate the Scenegraph from multiple independent threads without harming the data structure.
Another great feature is the addition of a reflection system to the data model which enables the programmer to write very generic tools since all objects in the Scenegraph can deliver information about themselves and their contained data at runtime.
OpenSG still has disadvantages the main problem is that its level of abstraction is very low so the programmer has to bring in a lot of computer graphics knowledge and can't relay on the frameworks convenience methods.

Avalon [1] is a VR/AR framework build on top of both OpenSG and VRML. It uses OpenSG to read in and display VRML files, but those VRML files are not standard VRML97 files. The Avalon project extended the VRML standard with a set of nodes that are needed for typical AR/VR systems including user input nodes (device handler, speech input etc.), a object to object collision detection, speech syntheses and advanced geometric objects.

Following table gives an overview over all inspected Frameworks:

	Java 3D	**VRML**	**OpenSG**	**Avalon**	**Coin3D**
Open source	yes	yes	yes	yes	yes
High level abstraction	limited	yes	limited	yes	yes
Threadsafe rendering	yes	no	yes	yes	yes
Threadsafe SG manipulations	yes	no	yes	yes	limited
Interaction Management	good	limited	limited	good	good
Authoring Tools	VRML import	yes	VRML import	no	yes

4.3 Open Inventor

Open Inventor is also a 3D description language it was originally developed by Silicon Graphics Inc. as closed source project. Since them several open source implementations have been developed and SGI made the original Inventor open source at the end of the nineties too, but decided to not continue the project instead the license was sold to TGS who since them continue the development of Inventor again as closed source project.

Parallel to the closed source efforts of TGS there is a second initiative which works on an Open Inventor implementation. The Coin3D project[3] achieved a 100% standard compliance to the SGI Inventor standard and works on compability to the TGS versions. Further they invented some additional features like import of CAD Models, a object to object collision detection and 100% VRML97 standard support.

Open Inventor is a object oriented 3D description language with the Scenegraph as core concept. Open Inventor nevertheless has got more to offer than VRML (and the other frameworks) in several ways. The main differences between VRML and Open Inventor are:

- Open Inventor was the superset of the original VRML(1) specification. Therefore it has got more available nodes to use in the Scenegraph (mostly for modeling dynamic behavior).
- In addition to the ASCII file format the Open Inventor standard delivers a C++ API with which the scene can be described programmatically and dynamic behavior can be modeled with all the power the C++ language offers.
- Open Inventor is an, as the name says, open standard and can be extended by custom nodes. Further the C++ API offers convenient ways to do this (Macros and helper functions [23]).

To sum this up one can say that Open Inventor is more powerful then the VRML standard and especially because of its great flexibility and extensibility we choose Open Inventor, and particular the Coin3D implementation, over available VRML Browsers because it is open source, stable and reliable, has got a very good documentation[2] and last but not least an active community. Also the possibility of programming OpenInventor in C++ makes it pretty easy to adapt a Open Inventor program to an DWARF service.

Even if Open Inventor/Coin3D support a sophisticated 3D framework there are still some problems. The main problems are:

- The Coin3D runtime engine is not thread safe which means that it is not possible to manipulate the Scenegraph from multiple independent threads. This leads to major problem since DWARF is by design multi-threaded.
- The Open Inventor standard was developed for desktop applications, therefore it only has got a 2D interaction management (keyboard, mouse). So interaction management for tracked objects and 6DOF devices has to be implemented.

4.3.1 Nodes

As stated above the core of Open Inventor is the Scenegraph which is assembled by nodes. The Open Inventor nodes can roughly be classified as follows:

Shape nodes basic geometric objects like cubes, spheres, cones, boxes etc.

Group nodes structure the Scenegraph and serve as container for one or more subgraphs.

Property nodes influence the way following `shape nodes` are rendered. Property nodes can influence for instance the material, size, position or transparency of an `shape node`.

Light and camera nodes are found a the very top of the Scenegraph they specify how the scene is lit and `camera nodes` function as virtual eye (viewpoint).

4.3.2 Interaction Management

In the following sections I want to describe some Open Inventor features of special interest in regards to interactivity which is very important for Augmented reality systems.

4.3.2.1 Actions

The core concept of Inventors dynamic model is a visitor pattern [14] called *action*. An action is applied to the root of a Scenegraph and then traverses all its children calling a distinct function on each of the nodes. Examples for Inventor Actions are:

SoGLRenderAction traverses the Scenegraph and accumulates the state, when the action reaches the leaf nodes it renders the Scenegraph to the display.

SoSearchAction searches for a specific node, a specific type of node etc.

4.3.2.2 Callback Nodes

Callback nodes allow actions to callback to the application passing the current traversal state. This means that whenever a callback node is traversed a specified function is called by the action. In this function, written by the programmer, arbitrary code can be executed for example some OpenGL calls could be made which is used to draw a video image to the back buffer in our case.

4.3.2.3 Sensors

Sensors are special database objects that monitor parts of the Scenegraph or just single nodes they respond whenever some of the observed values change. There are two types of sensors:

Data sensors invoke callback functions whenever a observed field changes.

Timer sensors invoke callback functions after a certain amount of time has changed.

Sensors can be used to animate the scene according to the change of certain values or just according to time. Further they can be used to avoid problems with threading, because with sensors changes on the Scenegraph can be made in 'safe' moments. I use Sensors to add and remove nodes from and to the Scenegraph.

4.3.2.4 Engines

Engines are also used to animate the scene. They allow the programmer to encapsulate motion and geometry into one single Scenegraph. Every Inventor object has got data fields. Those fields can be connected via engines to other objects' fields and thus control the behavior of the object. The Open Inventor engines form there own graph which lies on top of the regular Scenegraph including all connected nodes. Engines have got input values and build-in functions to calculate the output value, the input is collected by

traversing the superimposed engine graph.
Huge engine networks (meaning completely animated scenes) can be build using this technique.
Further engines can be used to constrain objects, for example camera rides could easily be achieved by connecting the position field of one object to the position field of the camera adding a certain offset so that the camera follows the other object.

4.4 Studierstube

The Studierstube Project [7] is a Augmented reality project at the technical university Vienna. Studierstube development started in 1996 it has brought up a variety of applications and scientific papers. The Studierstube framework is a Augmented reality framework with a strong focus on user interface research and mobile Augmented reality.
The Studierstube is basically an operating system for different Augmented reality systems. It is based on Open Inventor and a lot of Studierstube specific Inventor extensions. The applications themselves are mostly Open Inventor ASCII files that are executed by the Studierstube *Workspace* which is a 3D analogon to the 2D desktop[13].
The Studierstube is technical very advanced and fulfills most of the requirements of my work such as:

- line interleaved stereo
- anaglyphic stereo
- video see through
- +/- scene objects
- control objects

Still it suffers from one limitation that is caused by its design. All functionality provided by Studierstube is encapsulated in Open Inventor nodes even if the functionality hasn't got any relation to the graphical output subsystem (e.g. trackers). This concept is contrary to the DWARF paradigm of modularity.
Even if we can't use the Studierstube as out of the box solution for our problem we can reuse a lot of ideas and even code snippets from the Studierstube since all extensions made for the Studierstube have been written according to the Open Inventor extension guidelines [23]. This means that especially for the stereo modes and the video see through the Studierstube were my code base.

Chapter 5

Implementation

In this chapter I will describe how the requirements, gathered in 3, have been implemented. Also I will describe all classes, interfaces (and the usage of those) that are part of the developed component.
Throughout this chapter I will describe the 3D-viewer in increasing detail. Starting with the interfaces to the surrounding DWARF environment I will proceed through static class diagrams and descriptions. After that I will describe the dynamic behavior of the component using object models and sequence diagrams.

In figure 5.1 we can see how the 3D-viewer component fits into the DWARF user interface subsystem.

In addition to the *UserInterfaceController* shown in figure 5.1, the viewer can be controlled by the DWARF *Model* designed and described in [22] (see figure 5.2). The main difference is the character of the connection. one viewer can be controlled by several *UserInterfaceController* or just by a single *model*.

The connection between other DWARF services and the 3D-viewer is set up via *needs & abilities*. The 3D-viewer can be controlled with a set of predefined commands which are encapsulated in CORBA structured events [19]. The 3D-viewer has got (at minimum) the needs for *SceneData* and *UserAction*. those two events can contain the following commands:

SceneData contains commands to change the appearance of the scene graph.

- *CreateObject* creates a new scene graph object.
- *DeleteObject* deletes an existing scene graph object.
- *ReplaceScene* replaces the whole visible scene.
- *SuperImpose* superimposes the visible scene with 2D graphics.

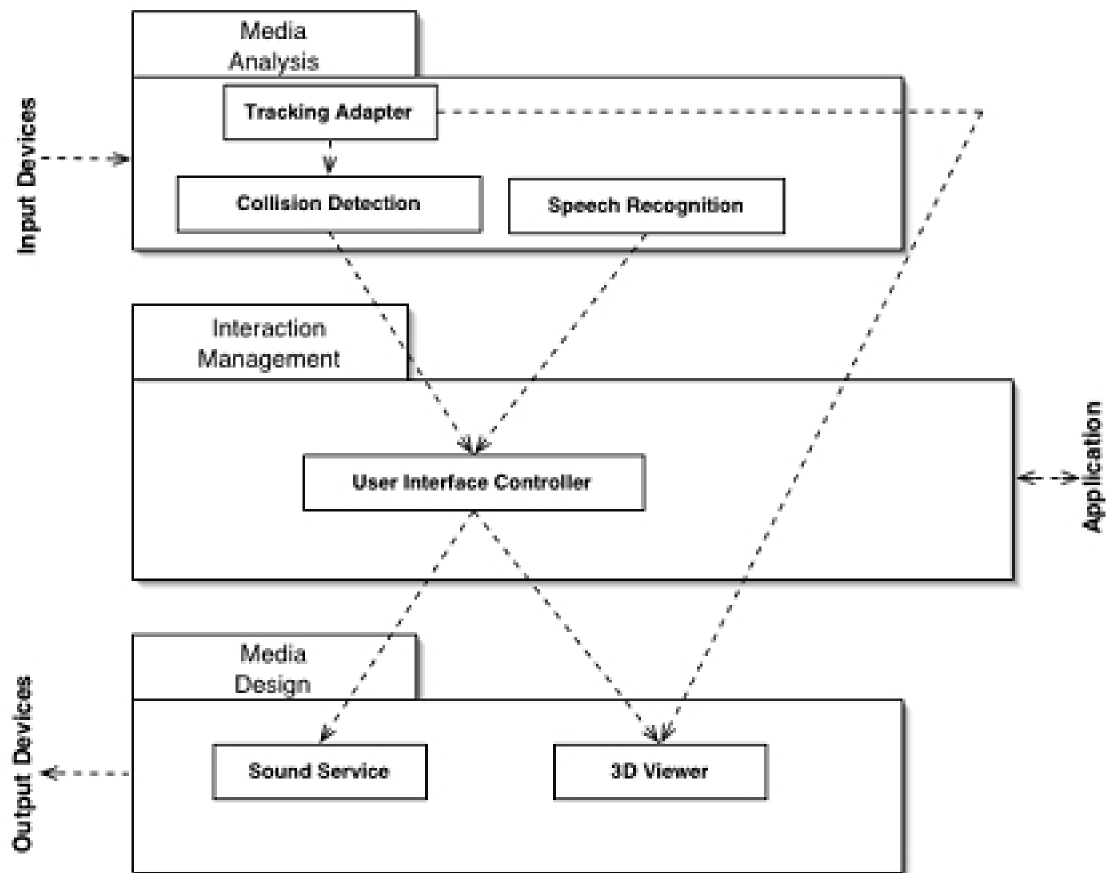


Figure 5.1: The DWARF user interface architecture

further the event contains several data fields which are used internal to execute those commands.

- *VirtualObjectId* the id for the scene graph object. This has to be unique throughout the application.
- *VirtualObjectParent* the parent object for a to-be created object.
- *newScene* an object as Open Inventor ASCII string that shall be added to the scene graph.

UserAction contains commands to change the behavior of single scene graph objects.

- *SelectVirtualObject* selects and accordingly connects a scene graph object to a real (tracked) object (respectively its *PoseData*)
- *DeselectVirtualObject* deselects a selected object.

further the event contains several data fields which are used internal to execute those commands.

- *VirtualObjectId* the virtual object that shall be selected/connected.
- *RealObjectId* the id of the real, tracked, object.

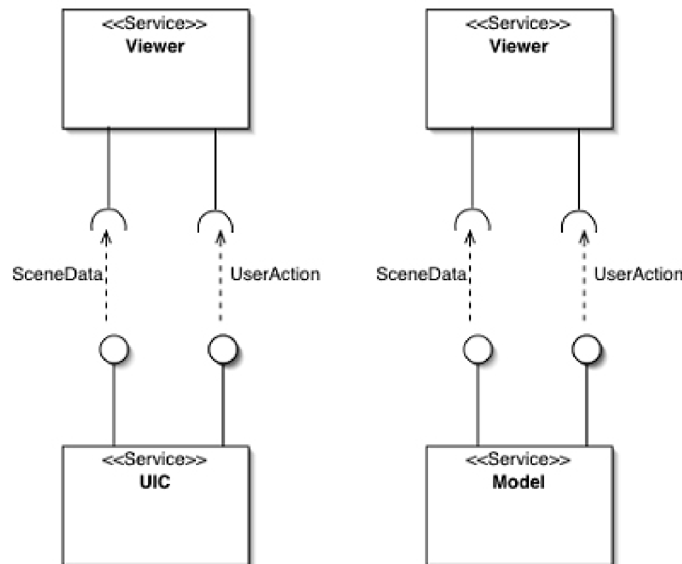


Figure 5.2: The interface of the viewer displayed in DWARF *needs & abilities*

Additionally the viewer can receive *PoseData* events to set the position of the viewpoint and arbitrary 3D objects.
The viewer can also receive video images via shared memory from the DWARF *VideoGrabber* service and display them in the background of the 3D scene.

5.1 Static Model

To understand the viewer component we have to take a look at the classes that aggregate to the needed functionality. An overview over all classes and there associations between each other is given in figure 5.3
To reduce the diagrams complexity I encapsulated several classes together into packages. Each of the new (sub-)packages contains classes with related functionality, since there is still a strong correlation between those packages I didn't break up the viewer component into smaller subsystems. The simplified architecture can be seen in (figure

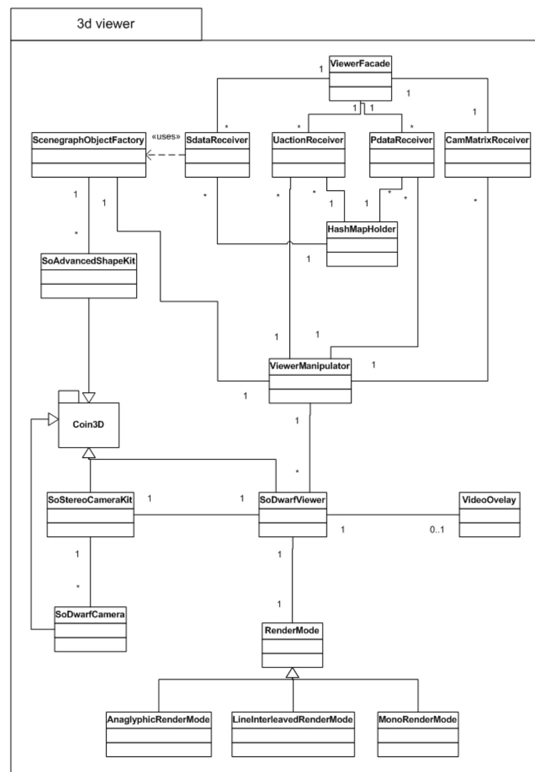


Figure 5.3: The class structure of the 3D-viewer.

5.4).

5.1.1 Communication and Event processing

The first package is the *ViewerFacade* package 5.5. It contains the *ViewerFacade* class itself and the *StructuredEventReceiver* package.
The *ViewerFacade* class is the only class that is seen from the outside system and all communication is done via this class. The *ViewerFacade* class is a singleton class [14], meaning that it has got only one instance for the whole runtime of the 3D-viewer. The *ViewerFacade* class implements the *BasicService_Shmem* interface and all the methods specified there. This means that the *ViewerFacade* class is a basic dwarf service and has got shared memory access for e.g. the video signal.

The second part of the *ViewerFacade* package is the *StructuredEventReceiver* subpackage. Because of performance reasons I chose to delegate the event-handling to separated classes. In this way no bottleneck occurs when a lot of incoming events arrive

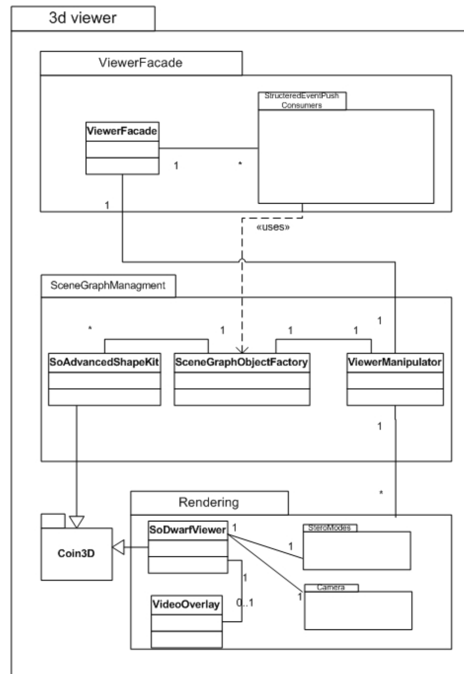


Figure 5.4: The simplified 3D-viewer architecture.

at the same point (in this case: in the same method). The *ViewerFacade* class instantiates dynamically one event consumer (see figure 5.6) per need in its service description which is passed as an argument in the *startService* method, called by the DWARF service manager. Each of the event consumers handles all incoming events of one need that means:

SdataReceivers process incoming *SceneData* 5 thus they create/delete scene graph objects, they replace the whole scene graph and they superimpose the scene with 2D information.

UactionReceivers process incoming *UserAction* thus they select or deselect scene graph objects.

PdataReceivers receive *PoseData* and set it as new position for selected scene graph objects or the viewpoint of the camera.

CammatrixReceivers receive calibration values computed by the SPAAM algorithm described in [26].

Since *SdataReceivers*, *UactionReceivers* and *PdataReceivers* need access to the same data mutual exclusion has to be guaranteed. Therefore the *HashMapHolder* class has been introduced to store references to all created scene graph objects in a thread safe

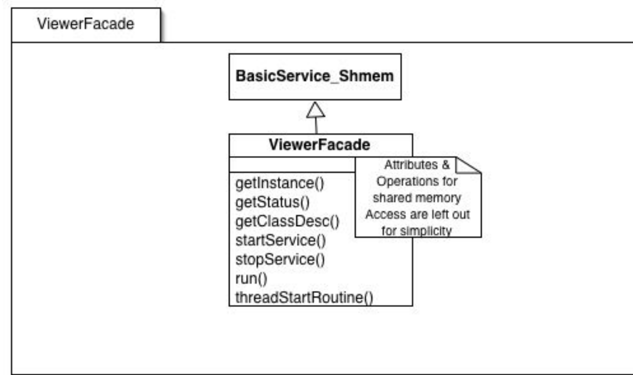


Figure 5.5: The *ViewerFacade* class, its attributes and operations.

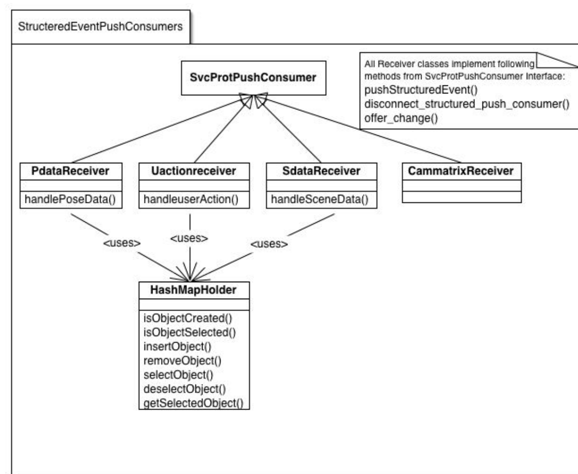


Figure 5.6: The structured event push consumers.

and persistent manner. The *HashMapHolder* class is using synchronized hash maps and accessory methods for this purpose. The *HashMapHolder* class also reduces the communication overhead because searching in the scene graph isn't necessary anymore.

5.1.2 Scenegraph Management

The next package is the *SceneGraphManagement* package (Figure 5.7) it contains three classes:

- *ViewerManipulator*
- *SceneGraphObjectFactory*
- *SoAdvancedShapeKit*

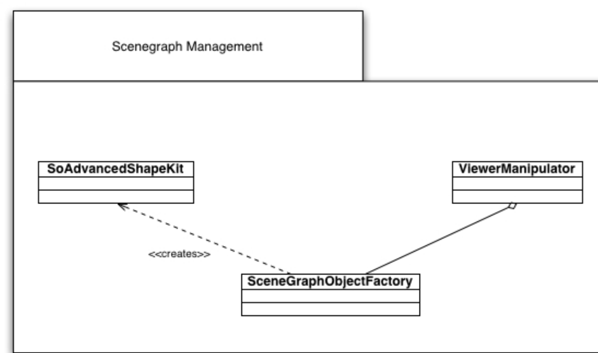


Figure 5.7: The*SceneGraphManagement* package.

The *ViewerManipulator* class is designed following the mediator pattern [14] that means that it encapsulates the interactions between different components of the system by preventing direct referrals between the single components. This has been done done out of two reasons. Firstly: the mediator pattern lets the programmer change the interactions in a complex system at one point without having to change all classes. Secondly: the threading problems of Open Inventor made it mandatory that changes on the scene graph are only done by one thread.
The *ViewerManipulator* class provides an interface for all manipulations that have to be done on the scene graph including addition/removal of objects change of position and calibrating the cameras etc.

The *SceneGraphObjectFactory* is a class designed after the principles of the factory pattern [14] which provides an interface to instantiate objects without specifying the exact class. In this case the *SceneGraphObjectFactory* creates scene graph objects of arbitrary geometry and encapsulates the geometry in a *SoAdvancedShapeKit*.

The *SoAdvancedShapeKit* class is a geometry container for arbitrary 3D objects. It is derived from the standard Open Inventor *SoShapeKit*[24][2] and provides custom

methods to control the geometries position in a simple manner (setPoseData() method).

5.1.3 Rendering

The last package is the *Rendering* package (Figure 5.8) it contains two classes and two sub-packages.

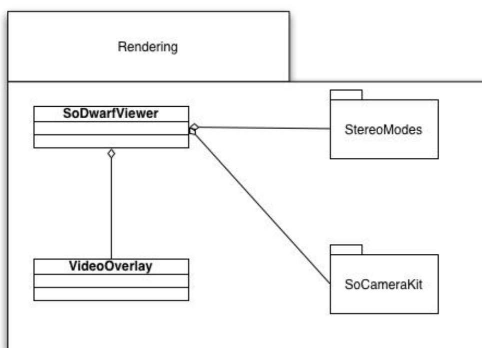


Figure 5.8: The*Rendering* package.

The main class is the *SoDwarfViewer* class it inherits from the Coin *SoQtExaminerViewer* class [2] and provides a variety of functionality. It's main feature is the rendering of the scene graph itself it uses therefore the mechanisms of the Coin3D [3] runtime engine.
The SoDwarfViewer is also a container for the scene graph that it renders. As seen in

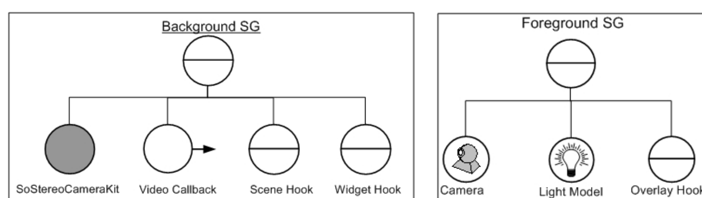


Figure 5.9: The initial scene graph stored in *SoDwarfViewer*.

figure 5.9 the initial scene graph consists of two parts. The background scene graph contains a *SoStereoCameraKit*, a callback node for video see through and two hooks (placeholders) for the actual scene and for widgets. The foreground scene graph is needed for overlaying the scene with 2D graphics it has got it's own camera (a *SoOrthographicCamera* [2]) a light model and again a hook for the scene that shall be displayed.

Further the *SoDwarfViewer* class provides several accessory methods to change the scene graph and to get information about the scene graphs state or elements of the scene graph (for more detailed documentation please refer to the doxygen documentation and see figure 5.10).
The different modes of the *SoDwarfViewer* class can be set with following command line

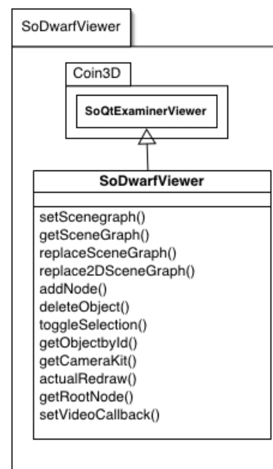


Figure 5.10: The *SoDwarfViewer* class.

parameters:

-Dstereomode (anaglyphic*lineinterleaved*none) (default: none).

-Dvideobackground video see-through yes or no (default: no).

-Dfullscreen fullscreen yes or no (default: no).

-Dresolutionx the size of the video image (default: 640).

-Dresolutiony the size of the video image (default: 480).

-Dmodel the file to be loaded as initial scene (default: none).

-Dbackgroundcolor 0=black, 1=white (default: 0).

The most interesting method in the *SoDwarfViewer* class is the *actualRedraw()* method it is called every time the whole scene graph has been traversed by the *SoGLRenderAction* and draws the scene to the screen.
The method is overridden from *SoQtExaminerViewer* to implement custom functionality. Since one can be sure that at this point no action is traversing the scene graph, this is the point where modifications to the scene graph can be done in a (thread-) safe way.
At the beginning of *actualRedraw()* all scheduled changes to the scene graph are actually executed (add/remove nodes, replace, superimpose).

```
if(!addQueue.empty()) addNodeInternal();
if(!hudQueue.empty()) replace2DSceneGraphInternal();
if(!sceneQueue.empty()) replaceSceneGraphInternal();
```

Further the *actualRedraw()* method takes care that the scene is really rendered in stereo. It makes the necessary calls to the stereo modes *selectBufferLeft* respectively *selectBufferRight* methods and draws the masked image into the back buffer.

```
//mask left image
getCameraKit()->selectCameraLeft();
myMode->selectBufferLeft();

//mask right image
getCameraKit()->selectCameraRight();
myMode->selectBufferRight();

//draw the image
SoGLRenderAction * glra->apply(root);
```

After that the method calls the parent method which causes a buffer swap so the image is drawn to the screen. Finally it schedules a retraversal of the scene graph.
The *actualRedraw()* method would be the starting point when other developers want to extend the *SoDwarfViewer* class with functionality that affects the scene graph.

The first sub-package is the *Cameras* sub-package. It contains two classes the:

- *SoStereoCameraKit* represents a node kit [2] with two *SoDwarfCameras* (one for the left eye and one for the right eye) mounted on a virtual rack that can be freely positioned. Using this kit, a viewer can manipulate (move) two cameras simultaneously, and switch cameras at will for rendering.
- *SoDwarfCamera* inherits from a *SoCamera*[2] and contains special methods to calibrate it according to the outcome of the SPAAM algorithm described in [26].

Both classes together form the virtual eye for the *SoDwarfViewer*. The *SoStereoCameraKit* has got methods to set the position of both cameras in one step and contains a switch to choose between the two cameras. In every rendering pass (in stereo mode) the scene is drawn twice once with the left camera and once with the right camera to produce a stereo image. The offset between the two cameras (eyes) is freely configurable.

The last sub-package is the *StereoModes* package it contains the stereo modes available for the *SoDwarfViewer* (figure 5.11).

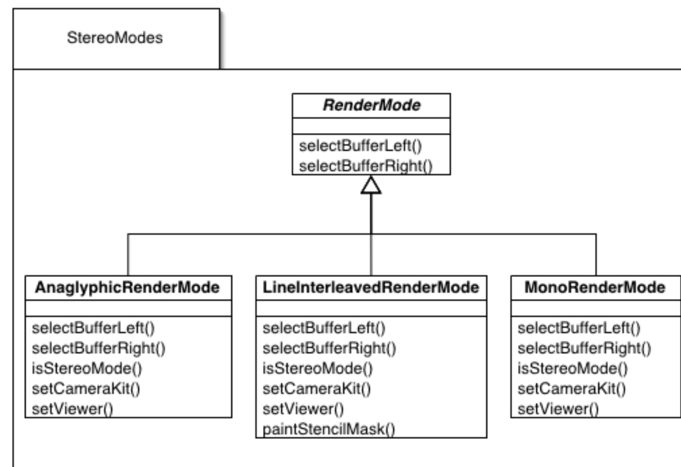


Figure 5.11: The Stereo modes available.

- *MonoRenderMode*
- *LineInterleavedRenderMode*
- *AnaglyphicRenderMode*

All three render modes inherit from the abstract *RenderMode* class, which makes it easy to add new render modes. All modes implement the methods *selectBufferRight()* and *selectBufferLeft()* which paint the appropriate mask for the right or left eye and put it over the image so that a correct stereo image is rendered.
In case of the line interleaved mode for the one eye all lines with even linenumbers are stamped out and for the other eye all lines with odd linenumbers are stamped out.
The anaglyphic render mode puts a red color mask over the image for the first eye and a cyan colored mask over the image for the second eye.
The mono mode doesn't do any modifications on the images.

The last part of the *Rendering* package is the *VideoOverlay* class. The most important method is the *drawVideoImage()* method which is called during every render pass (utilizing Inventors callback mechanism). It draws the current video image (which is received via shared memory in *ViewerFacade*) to the background buffer.

5.2 Dynamic Models

Now that the classes which form the 3D-viewer component are known I will describe how the instances of those behave dynamically during usage.

5.2.1 Object Model

In figure 5.13 a object model of the viewer is shown. The viewer instance is setup with anaglyphic stereo (see 5.1.3). Each of the different *StructuredEventPushConsumers* belongs to one need (with *PushConsumer* connector) declared in the service description (see figure 5.12).

```
<service name="Viewer" startCommand="Viewer
 -Dstereomode=anaglyphic -Dservicename=Viewer">

  <need name="6DObject" type="PoseData" predicate="(&amp;(ThingType=6DObj
    <connector protocol="PushConsumer"/>
  </need>
  <need name="viewpoint" type="PoseData" predicate="(&amp;(ThingType=View
    <connector protocol="PushConsumer"/>
  </need>
  <need name="ManualTracker" type="PoseData" predicate="(&amp;(RealData=f
    <connector protocol="PushConsumer"/>
  </need>
  <need name="Scene" type="SceneData" predicate="(poseChanges=false)">
    <connector protocol="PushConsumer"/>
  </need>
  <need name="Action" type="UserAction">
    <connector protocol="PushConsumer"/>
  </need>
  <need name="Camera" type="CameraMatrix">
    <connector protocol="PushConsumer"/>
  </need>
  <need name="SheepPose" type="PoseData" predicate="(ThingType=Sheep)">
    <connector protocol="PushConsumer"/>
  </need>
  <need name="SheepScene" type="SceneData">
    <connector protocol="PushConsumer"/>
  </need>
</service>
```

Figure 5.12: The service description for the 3D-Viewer in XML

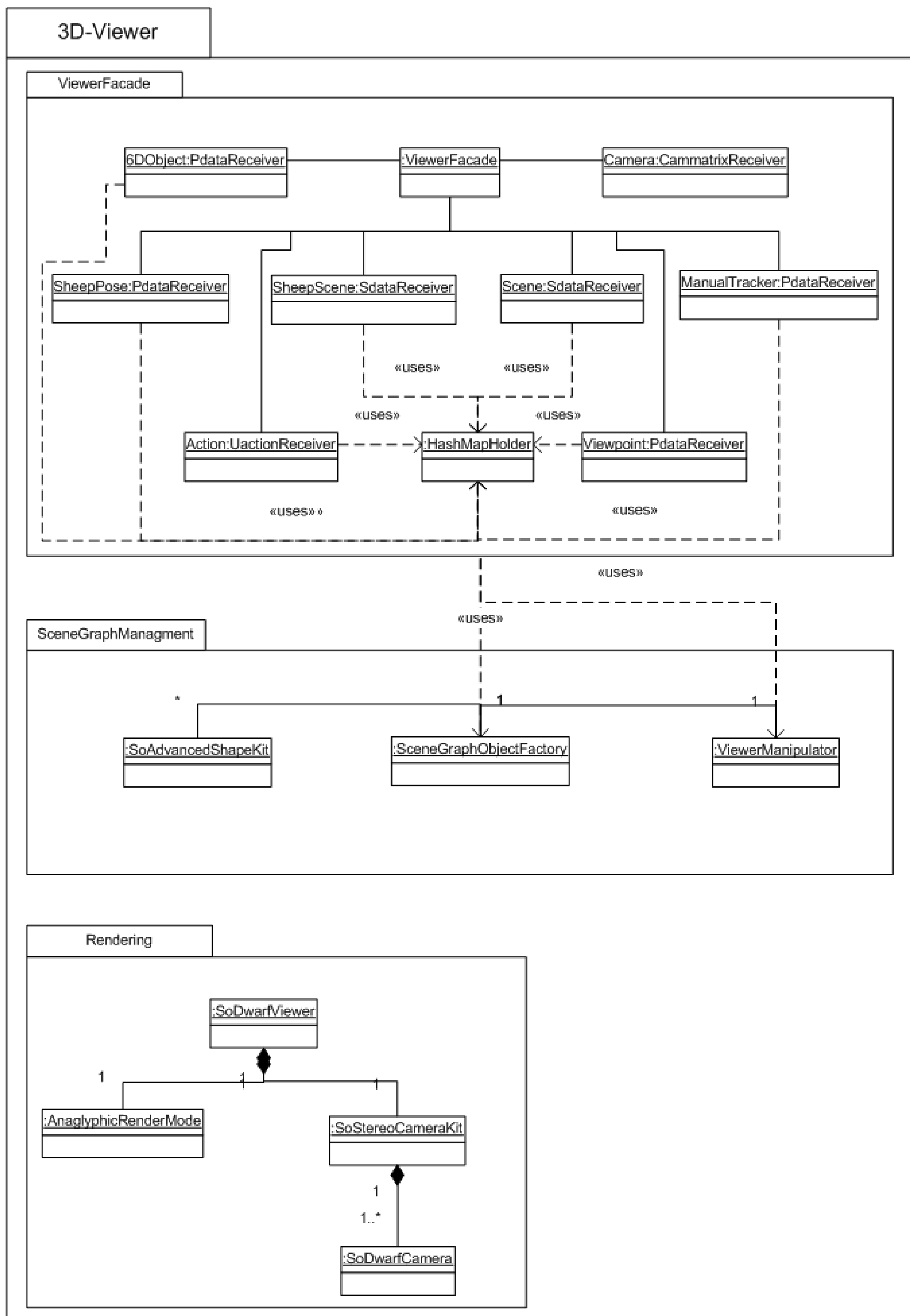


Figure 5.13: Object model resulting from service description given in 5.12.

5.2.2 Sequence Diagrams

In this section I will show by means of sequence diagrams how the objects communicate between each other to fulfill the use cases described in 3.2.

The most important thing for a Augmented reality viewer component is that the user can actually see and inspect the scene. Therefore the virtual viewpoint has to be adjusted to the users head position. In figure 5.14 is illustrated how incoming *PoseData* is handled by the viewer. The incoming *PoseData* is delegated to the *PdataReceiver* which identifies the object that shall be repositioned, in this case the camera, and finally the *ViewerManipulator* sets the new values utilizing the *setPoseData()* method of the *SoStereoCameraKit*. All objects that have been created have this method so setting *PoseData* for arbitrary objects works analog.

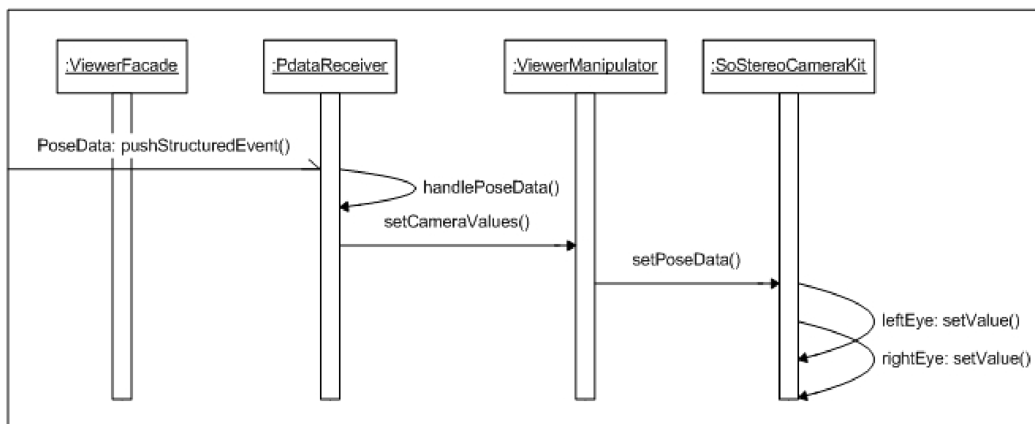


Figure 5.14: Setting the virtual viewpoint.

The next diagram (5.15) illustrates how new objects are created and inserted into the scene. When the user gives a command to insert a new object e.g. via speech input or pressing a button (this is application dependent and not part of the 3D-viewer component) a *SceneData* event is generated and sent to the 3D-viewer.
The incoming event is delegated to the *SdataReceiver* which utilizes the *SceneGraphObjectFactory* (passing through the Open Inventor ASCII description of the geometry) to create an new *SoAdvancedShapeKit* encapsulating the geometry. A reference to the scene graph object is stored in a hash map by the *HashMapHolder*.
The *ViewerManipulator* calls the *addNode* method of *SoDwarfViewer* which finally adds the new node to the scene graph at the beginning of the *actualRedraw()* method (this eliminates threading problems).
Deleting Nodes works analog.

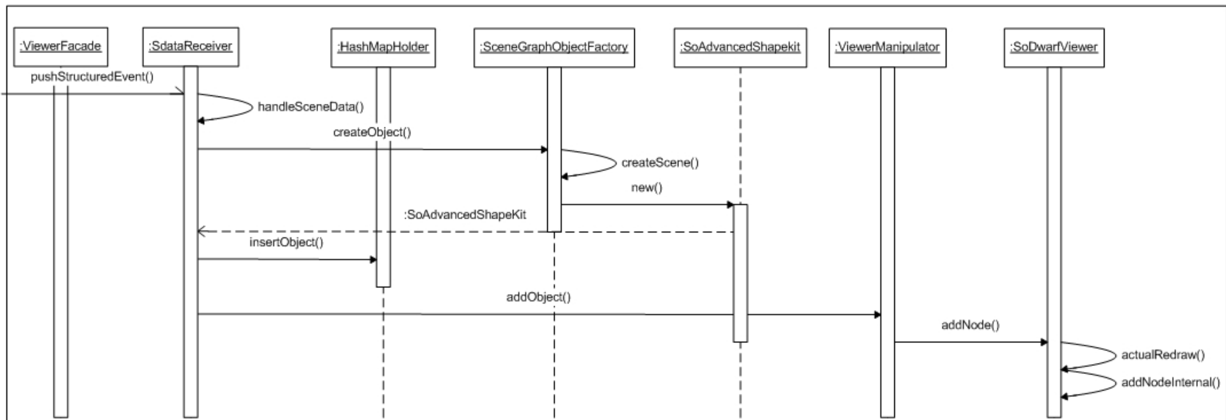


Figure 5.15: Adding a 3D object to the scene graph.

The last diagram (5.16) shows how the whole scene graph is replaced with a new one. Note that the actual replacement is, again, conducted in the *actualRedraw()* method. Setting the initial scene graph and superimposing the scene graph works in the same manner.

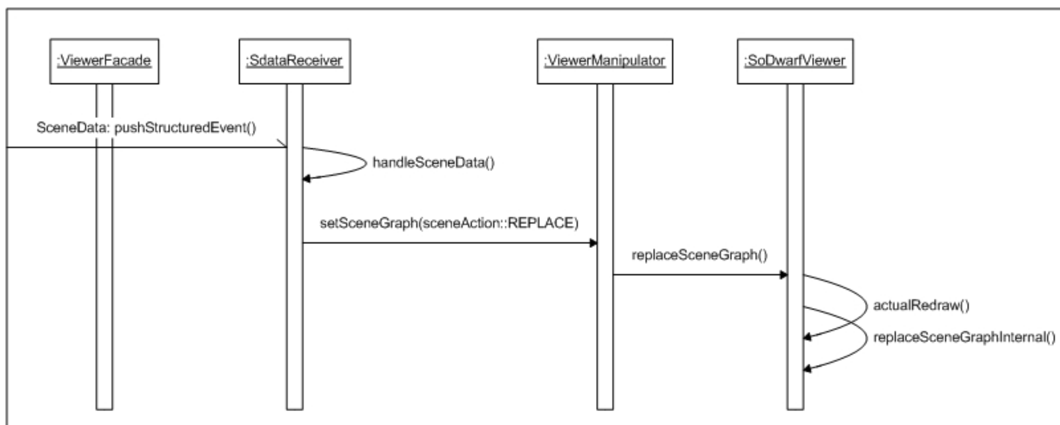


Figure 5.16: Replacing the whole scene graph.

5.3 Benchmarks

Concluding the implementation the new 3D-viewer component has been benchmarked. Since the 3D-viewer is used for Augmented reality systems, which means that rather small models are rendered, I didn't Benchmark the component with huge models and

large scale data sets, this would only show how powerful the Coin3D runtime engine is, which I can't influence.
Instead I did tests concerning the interactive changes made on the Scenegraph. Those changes are caused by incoming events from the other DWARF subsystems. Data with a high update rate (e.g. (PoseData)) was of special interest because they are critical to the render performance.
To measure how the new 3D-viewer component handles lots of incoming *PoseData* events I ran two tests. In both tests I displayed the landscape used in the Sheep demo [20] and started as many sheep as possible. The stop criterium was either the minimum frame rate of 25 fps (see 3.3.2) or the capabilities of the DWARF middleware. The tests have been run on a Pentium 4 2.5Ghz laptop equipped with 512MB of ram and a Ati Radeon 9000 mobility graphics adapter.

- In test one the update rate of each sheep was 2 Hz (2 *PoseData* events per second). During the whole benchmark the viewpoint was set with 20 Hz. The maximum frame rate was 220 fps without any sheep running. The minimum frame rate was 36 fps with 15 sheep running and a total of 50 Hz incoming *PoseData* events (see figure 5.17). It was not possible to start more sheep because the *servicemanager* couldn't handle more connections (the sheep form a complete graph, giving a total of (nr. sheep=n) $\binom{n}{2}$ connections and that results in a huge amount of threads). The frame rate might go down a little bit when the viewpoint is changed rapidly (e.g. while abruptly turning the head).

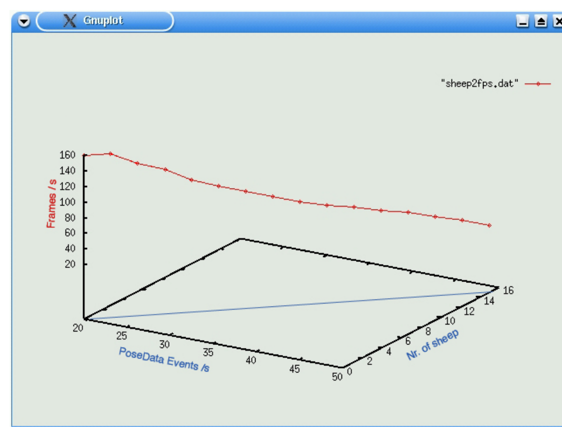


Figure 5.17: Frames per second according to the nr. sheep displayed in the viewer (2Hz update rate each sheep).

- In test two I used the same setup but changed each sheep update rate to 10 Hz which makes them move smooth. The maximum framerate was again 220 fps with no sheep and the minimum was 71 fps with 10 sheep and a total of 120 Hz update rate (see figure 5.18). The 11th sheep caused the notification service (part of the

DWARF middleware) to crash, I assume that this was caused by the high update rate of 120 Hz but more research is necessary here.

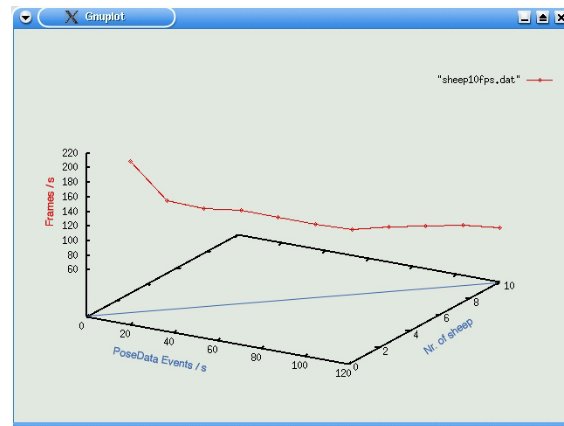


Figure 5.18: Frames per second according to the nr. sheep displayed in the viewer (10Hz update rate each sheep).

Both tests have shown that the 3D-viewer is capable of handling lots of incoming events in a performant way and is usable for Augmented reality systems. (This was also proved by the ARCHIE demonstration at the end of the project.)
As result of both tests we can say that the 3D-viewer component can handle fewer connections with a higher update rate better then a lot of connections with a smaller update rate. This could be improved by changing the granularity of the (StructuredEventPushConsumers) (see 5.1.1). In the current implementation for each (need) a corresponding (StructuredEventPushConsumer) is instantiated. This could be changed so that a (StructuredEventPushConsumer) is instantiated for each supplier of (StructuredEvents) dynamically.

Chapter 6

Conclusion

6.1 Results

In this document I have shown why a new 3D-viewer component was needed for DWARF and which requirements have to be fulfilled by such a component (Chapter 1 through Chapter 3).
In Chapter 4 I have inspected and explained related work and familiar frameworks.
In Chapter 5 I have shown how the requirements and ideas gathered in the previous chapters have been implemented. In Chapter 5 I have also described how the 3D-viewer component interacts with the surrounding DWARF environment and how to use it in Augmented reality systems.

Finally the 3D-viewer component itself is a product of my work within the ARCHIE project. The component has reached a relative stable state so it was not only used in the ARCHIE project but has become the standard viewer for the DWARF project.
It is reused in several projects that are currently developed at the chair for applied software engineering (Technische Universität München). Among them are:

- ARCHIE (finished may 2003)
- SEP: Empirical estimation of tracking ranges and application thereof for smooth transition between two tracking devices (Sven Hennauer).
- SEP: Automatic Layout of GUI-Elements for Augmented Reality (Michael Riedel)
- HeARt (Heart surgery Enhanced by Augmented Reality Techniques)
- BAR (BMW Augmented Reality)
- Sheep v2.0

6.2 Lessons Learned

I have learned a lot during the ARCHIE project. Since I had a good understanding of the DWARF framework in advance I could concentrate on increasing my skills in several technical aspects. I have learned C++ as new programming language and I have become familiar with the Coin3D API as well with modeling in VRML.
Further I've learned a lot about project oriented work in a team, which requires a lot of management and communication among the team members.
The outcome of my work wouldn't be as good without all the discussions with my team members and coaches which lead me to a lot of good ideas and solutions.

6.3 Future Work

In this section I want to point out what has to be done yet, to improve the 3D-viewer component.

6.3.1 Technical Improvements

I have mentioned above that the component has become relative stable but still some problems exist.
Among them is the error handling, which has to be improved. So far the error handling only prevents the component from crashing but doesn't prevent the application from entering illegal or senseless states. A mechanism that handles errors in a more intelligent way is needed. Eventually a mechanism could be implemented that requires the interaction of the user and out of that interactions the system could learn how to react in certain situations.
A rather easy problem to fix is the naming of special parts of the component due to early decisions in the design process of the ARCHIE system.
Firstly the *UserAction* event should be renamed or at least marked as deprecated (for backwards compability with other ARCHIE components) and replaced by a more appropriate name.
It would be even better if the interface between the 3D-viewer component and the other DWARF components would be encapsulated in a command structure according to the command pattern [14].
Secondly *SceneGraphManager* would be the more appropriate name for the *ViewerManipulator*5.1.2 class.

6.3.2 Functional Improvements

The next mayor field for future work would be the implementation of a widget system to manipulate objects in the scene indirectly, e.g. a slider for changing the color of objects, and of course the implementation of a basic set of widgets that can be used in different DWARF applications.

Very familiar to the above problem is the development of a interface, or rather guidelines, to extend the viewers functionality in a natural way (Open Inventor style).

6.3.3 Further Leading Ideas

Finally I want to point out what ideas came to my mind that could be interesting to investigate or implement in the future, but no work has been done yet.
First of all one could think about more stereo modes for different hardware e.g. field sequential stereo mode for Sony's Glasstron HMD [6] and a variety of stereo beamers as such hardware becomes available at the chair for applied software engineering.
For a even higher degree of realism a occlusion handling mechanism (using depth maps) has to be implemented so that virtual objects can be occluded by real objects, e.g. the users hand.
Another very interesting field would be a 'context aware' viewer component that means, that the scene displayed to the user changes according to the environment conditions and according to the users actions and position. For example a map, for navigation, could be displayed when the user looks down and when the user looks up again the original scene would be displayed.
At last one could have a deeper look at the frameworks described in 4.1 and 4.4 and find out which of the (custom) nodes could be of interest for the DWARF project and port those to the DWARF 3D-viewer component.

Bibliography

[1] *Avalon project homepage*. http://www.zgdv.de/avalon/.

[2] *Coin 3D documentation homepage*. http://doc.coin3d.org.

[3] *Coin 3D Project Homepage*. http://coin3d.org.

[4] *Java3D homepage*. http://java.sun.com/products/java-media/3D/.

[5] *OpenSG homepage*. http://www.opensg.org/.

[6] *Sony Cooperation Homepage*. http://www.sony.com.

[7] *Studierstube Project Homepage*. http://studierstube.org.

[8] *Virtual I/O Homepage*. http://www.i-glasses.com.

[9] *DWARF Project Homepage*. http://www.augmentedreality.de.

[10] *VRML97 specification*. http://www.vrml.org/technicalinfo/specification/vrml97/index.htm.

[11] M. BAUER, B. BRUEGGE, G. KLINKER, A. MACWILLIAMS, T. REICHER, S. RISS, C. SANDOR, and M. WAGNER, *Design of a Component-based Augmented Reality Framework*, in Proceedings of the IEEE and ACM International Symposium on Augmented Reality (ISAR 2001), 2001.

[12] B. BRÜGGE and A. H. DUTOIT, *Object-Oriented Software Engineering. Conquering Complex and Changing Systems*, Prentice Hall, Upper Saddle River, NJ, 2000.

[13] D.SCHMALSTIEG, A. FUHRMANN, G. HESINA, Z. ARI, L. ENCARNAC, A. GERVAUTZ, and W. PURGATHOFER, *The Studierstube Augmented Reality Project*, 2000.

[14] E. GAMMA, R. HELM, R. JOHNSON, and J. VLISSIDES, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Publishing Company, 1994.

[15] J. HARTMAN and J. WERNECKE, *The VRML 2.0 Handbook*, Silicon Graphics, Inc., 1996.

[16] C. KULAS, *Usability Engineering for Ubiquitous Computing*, Master's thesis, Technische Universität München, 2003.

[17] M. KURZAK, *Architecture and Urban Planning*, Master's thesis, Universität Stuttgart, 2003.

[18] M.-J. LEE, *Authoring of 3D User Interfaces for Applictions with DWARF*. Bachelorarbeit, Technische Universität München, 2002.

[19] A. MACWILLIAMS, *Using Ad-Hoc Services for Mobile Augmented Reality Systems*, Master's thesis, Technische Universität München, 2001.

[20] A. MACWILLIAMS, C. SANDOR, M. WAGNER, M. BAUER, G. KLINKER, and B. BRÜGGE, *Herding Sheep: Live System Development for Distributed Augmented Reality*, in Proceedings of ISMAR 2003, 2003.

[21] C. SANDOR, A. MACWILLIAMS, M. WAGNER, M. BAUER, and G. KLINKER, *SHEEP: The Shared Environment Entertainment Pasture*, in EEE and ACM International Symposium on Mixed and Augmented Reality ISMAR 2002, 2002.

[22] M. TÖNNIS, *Data Management for AR Applications*, Master's thesis, Technische Universität München, 2003.

[23] J. WERNECKE, *Extending OpenInventor*, Silicon Graphics, Inc., 1994.

[24] J. WERNECKE, *The Inventor Mentor, Programming object oriented 3D Graphics*, Addison Wesley Publishing Company, 1994.

[25] B. ZAUN, *A Bluetooth Communications Service for DWARF*. Systementwicklungsprojekt, Technische Universität München, 2000.

[26] B. ZAUN, *Calibration of Virtual Cameras for AR*, Master's thesis, Technische Universität München, 2003.