Space Trash

–

Development of a Networked Immersive Virtual Reality Installation

Technical Report

TR-GUP-01-09

by

Christoph Anthes, Phillip Aumayr, Clemens Birklbauer, Roland Hackl, Marlene Hochrieser, Roland Hopferwieser, Roland Landertshamer, Bernhard Lehner, Marina Lenger, Martin Lenzelbauer, Clemens Mock, Simon Opelt, Christoph Payrhuber, Mika Satomi, Stefan Simmer, Georg Stevenson, and Alexander Wilhelm

Institut für Graphische und Parallele Datenverarbeitung
Johannes Kepler Universität Linz

Linz, June 2009

Abstract

Virtual Reality (VR) and new media art are two topics that have always been closely intertwined. VR as an ideal tool to stimulate the user's sense in order to create immersion and presence, and new media art using this technology form a natural symbiosis in order to foster creativity and circulate ideas. By combining three different groups from the areas of VR software development, interface design and 3D modeling a development process was established in order to design and create a VR application using novel input devices.
The *Space Trash* game was developed to demonstrate the capabilities of VR technology, software aspects of Networked Virtual Environments (NVEs), design considerations of novel input devices as well as modeling approaches for real-time 3D objects. In this report the development of this interactive stereoscopic installation, which combines aspects from game, interface and visual design as well as VR software engineering, is presented.
This document will in detail describe the previously mentioned approaches and design aspects.

Acknowledgments

> These are the credits. We call them credits because we can't really afford to pay anybody until more people buy the game.
>
> Roger Wilco - Space Quest

We would like to thank all the people involved with this project, who have spent a significant amount of time and effort to make it come true.
Besides the authors many others were involved providing interface parts, supporting the application development and installation.

Contents

Chapter 1

Introduction

> ... to boldly go where no man has gone before.
>
> James Tiberius Kirk - Star Trek

Space Trash is a fully immersive Virtual Reality (VR) installation making use of dedicated physical interfaces. It was initially developed for demonstration purposes at the European Researchers' Night 2008. With *Space Trash* two teams of players compete by collecting satellite and spaceship parts in earth's orbit.
This report describes the different aspects of the development of the application. The main focus is set on the software architecture and the different software components which are created for the game.

1.1 Motivation

The task of this project was to develop a demonstration application for a public event, the European Researchers' Night 2008. The application had to illustrate the research in the area of VR of the Institute of Graphics and Parallel Processing (GUP) at the Johannes Kepler University (JKU) and physical interfaces at the Interface Culture Department (IC) of the University of Art and Industrial Design Linz (UFG). Past cooperations between the JKU, the UFG and an independent design company the Visioneers have been very fruitful as for example during the development of the Net'o'drom [AWL+07] application which was created for the Night of Science 2005.
The initial idea was to provide an installation, which is interactive, immersive, entertaining, technologically challenging and had a topic that was important for all developers.
Early discussions in finding such a topic ranged from areas of simple puzzle games of the field of casual gaming over to political topics like the "surveillance society". Most of the preliminary ideas turned out to be either of too little content or were too difficult to cover in the area of ludicity or it was to complicated to transport the message with the given constraints.
The topic of space debris seemed to be currently a very hot topic. Furthermore, gaming ideas

could be quickly developed in this setting and it was obvious that it could work ideally with stereoscopic displays.

1.2 Approach

The application and installation design as well as the development was performed collaboratively by three independent groups from the areas of interface design, graphical design and computer science in several stages.
Initial discussions were held regularly in order to find a topic and create a realizable concept, considering the means of time, budget, and effort. The topic of space trash has been found ideal for such a presentation since it sufficed the creators demands for having a real world relation and meaning. It was easily possible to find an entertaining gameplay and to create interfaces which could support this gameplay. Last but not least the topic could be realized visually appealing especially in the context stereoscopic multi-display systems.
In order to implement the application a clear definition of the game mechanics had to be created first. The device possibilities had to be evaluated and the software architecture had to be sketched.
After defining initial design questions the three teams started working on their specific topics but were communicating in between to close unsolved issues.
Communication protocols between the application and the devices were drafted. The visual design was sketched out and the architecture connecting the identified application components was specified. These three aspects will be briefly described in the following.
The interface design for the *Space Trash* installation follows the approach to develop interfaces which are already well known to the user and produce a direct association between the way how to use them and the reaction of the application. These devices had to fit the application topic as well, therefore they were mainly built out of trash pieces and previously owned parts.
The graphical design of the application focuses on two different aspects the narration and the setting of the game as well as user guidance to support the gameplay. Many 3D models representing the trash, the satellites and the users' space ships had to be designed and created. To support the atmosphere of the game a soundtrack was composed and sound effects were recorded and post-processed.
The main focus of this document – the software implementation – was based on the *inVRs* framework [AV06] which allows the easy creation of Networked Virtual Environments (NVEs) and uses OpenSG [Rei02] as a scene graph for the abstraction of low-level OpenGL code. The game logic had to be developed fully from scratch and additional components, like simulation of a grabbing arm, interaction and navigation mechanisms, a camera path, a Head-Up Display (HUD), and graphical effects were created to support the game mechanisms and the visual design. One additional aspect was the physics simulation of rigid bodies which is used to create the feeling of a vivid virtual world and support the game mechanisms in the areas of navigation and collisions.
The application itself can be considered highly collaborative although the gameplay itself is competitive. Two sites with a team of three players at each site are interconnected in this game. A

really novel aspect is the collaboration inside the teams which is needed to successfully beat the other team. This collaborative aspect is furthermore supported by the design of the physical interfaces.
A short version of this document has been previously published at the VRIC conference in Laval [ASW+09] where the application itself has been presented at the Laval ReVolution festival to a large public audience.

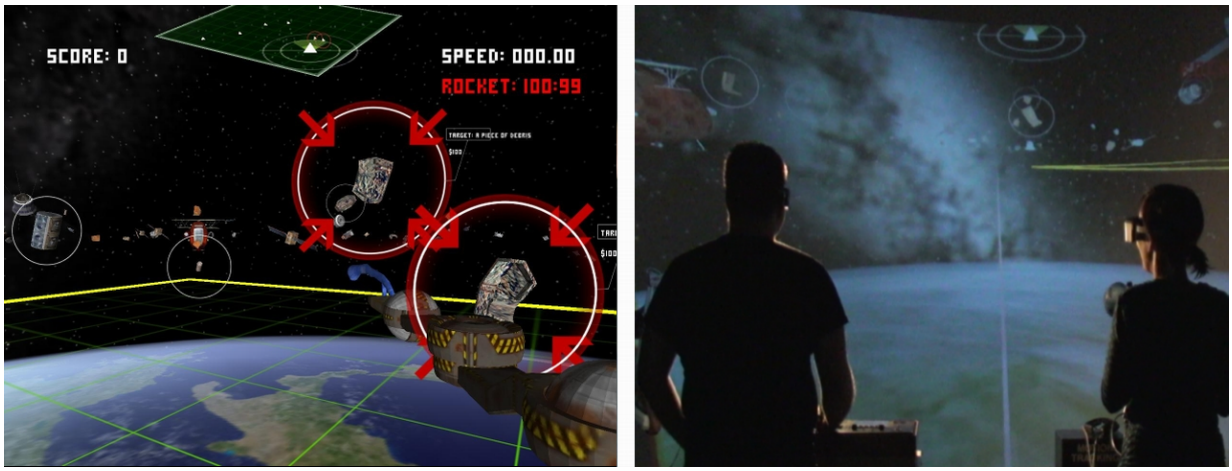

Figure 1.1: The *Space Trash* Installation

Figure 1.1 shows a screenshot from the application displaying the earth, trash pieces, and a grabbing arm on the left side as well as two user playing the game at the European Researchers' Night 2008 incorporating a stereoscopic multi-display installation on the right side.

1.3 Outline

The report is structured in the following sections:

- Chapter 2 - Game Design
  Concepts of the game play, the rules and the initial design are provided. The storyline introducing two groups of characters is given.
- Chapter 3 - Graphical Design
  The ideas behind the visual design are illustrated looking at both aspects the narration of the game as well as the user guidance.
- Chapter 4 - Input Devices
  A set of physical interfaces was created for the application. The design and the mechanics of these devices are explained in this chapter.
- Chapter 5 - Software Architecture
  The software architecture of *Space Trash* is presented. As a framework on which the application is based upon *inVRs* is used which will be explained as well in its main components.

- Chapter 6 - Gameplay Components
  Many different components are needed to support the gameplay and the user. These components are explained on a technical level.

- Chapter 7 - Animating Space Trash
  A variety of dynamic elements are available in the application. A pre-defined camera sequence, animation sequences of satellites and the simulation of the grabbing arm are introduced.

- Chapter 8 - Graphical Effects
  To provide a more vivid virtual world graphical effects had to be developed. Thus, this chapter provides an overview on shaders, reflection maps and shadow mapping.

- Chapter 9 - Audio
  The soundtrack of the application which is used to support the atmosphere as well as the sound effects providing sonification of the gameplay are explained.

- Chapter 10 - Conclusions
  This chapter concludes the report by providing an overview on the installations at the European Researchers' Night 2008. A brief outlook on future work is given.

Chapter 2

Game Design

> Scanning quadrant. Lieutenant, I'm picking up metal fragments ahead. Alloy structure indicates it's from a Confederation vessel.
>
> Merlin - Wing Commander

The game strategy including the story line, the game mechanics as well as the player interaction scheme was discussed at the first stage of the project. Due to the context of the presentation, some preliminary requirements were given. The playing time had to be less than 10 minutes, it should present the technology of a two site network communication, and the game control scheme must be intuitive for non-gamers of all age groups.

Game Mechanics	Challenge
	Reward
	Obstacles
	Interface
	Competition type
Visual Style	Style
	Contrast
	Information Layout
	Motion
	Sound/Audio
Story Line	Drama
	Character
	Motivation (Motivation of the character)
	Identification (Role of the player)

Table 2.1: Video Game Analysis by Kirschner

A video game analysis table (see Table 2.1) created by Friedrich Kirschner [Kir08] was used in

order to analyze the planned game idea on paper before stepping into the prototyping phase. The considerations made during the decision process as well as the final planning of each category are explained in the following sections.

2.1 Game Mechanics

The game is designed as a two site competitive game using network communication. The competition is focused on the speed and timing of the game play rather than strategy or skill. Since most of the participants are playing only once during the presentation, and playing time is restricted to 10 minutes, the player's learning curve in the aspect of skill is limited. Therefore the interaction is planed as micro-level interaction scheme with moment-to-moment choices of players rather than macro-level interaction scheme concentrating on the accumulation of micro-level choices. Such a macro-level approach requires a significantly larger curve of experience [SZ03].
The game control, meaning the direction of the ship, the speed of the ship as well as the aiming of the grabbing arm, is assigned to three individual physical interfaces controlled by three players at each site. The players' choices are made on a micro-level, allowing the non-expert players to manage the game play. The movement of the space ship is based on an accumulation of three players' actions, which requires choices made in macro-level game play through communication and collaboration in between the users.
The reward is given as money (score) as the players collect the space trash. Each site is represented as space ship in the game, which can bump into the other players' ship in order to disturb a controlled movement.

2.2 Visual Style

The application is displayed using stereoscopic images. To maximize the 3D effect, the earth with a realistic texture mapping is placed on the bottom of the displayed environment, and the rest of the background is kept as a dark deep space. The overall visual style of the environment and obstacles is kept realistic, while the player representations (space ships) style is designed with a comic style texture mapping. The perspective is set from a first person point of view to simulate the space ship's window. Additional game control information such as collectible space debris, the price of each trash piece and the playing area is superimposed in form of a Head-Up Display (HUD).

2.3 Storyline

The storyline chooses the setting of the fictional world that the users experience during the game. We have chosen the topic of space trash, which is a constantly increasing problem in space exploration. It also implies how we are dealing with our environment. Our aim is not to educate the player through game play, but to raise awareness for existing social issues. Following is the summary of the storyline:

10 years from now, the earth orbit is filled with space debris produced from previous space excursions. There is a growing necessity to collect the space trash and create room for the launch of new satellites. Many freelancing space trash collectors are making contracts with big satellite corporations to take this dangerous and high rewarded job. Now, there is a competition between collectors to gather as much trash in a limited flight time in order to make more money.

The character background describes the two different teams:

Three old retired astronauts from NASA, Russia and China, who lost their money in Las Vegas and are looking forward to earn the peaceful retired life in Florida. Three hackers flying in a self-made spaceship. Everything on the ship is made of used electronics parts. Their aim is to launch their DIY satellite for the "FREE-NETWORK".

Figure 2.1 shows an initial concept drawing of the group of three hackers which control a space ship.

Figure 2.1: Concept Drawing of the Hackers

The introduced story describes coherent worlds that refer to the existing social problems such as pollution in space, thus the game can be considered as a coherent world game [Juu05] that simulates the real world partly identifying the players as protagonists. This storyline and the character background are introduced through the aesthetic design of the 3D graphics and the physical interfaces.

2.4 Summary

This chapter has briefly introduced the game design concepts of *Space Trash*. The game mechanics, the visual style as well as the background story were described. Reasons for these design decisions were given.

Chapter 3

Graphical Design

> Sir, Dr. Evil's not bluffing. One of our satellites is falling out of orbit.
>
> Johnson - Austin Powers

The design of the game can be categorized into two functional groups. First the storytelling component of the narrative design and second the components which are used for user support and guidance. In a successful composition both aspects are skillfully intertwined that they can not be separated on first sight.

3.1 Narrative Design

The goal of the narrative design is to carry all aspects of the background story which cannot be explained via text or speech. The effect of the artistic elements unfolds solely through the design of the objects (e.g. the space ships), with the use of 3D sculptural language and the formalization of the materiality with the help of the used textures.
The basic assumption with every narrative design is that the impression of a design expresses itself in a sublime manner and is perceived parenthetically by the observer. The goal of the narrative design is the authentication of the presented story, in order to keep it as plausible and consistent as possible. The designer identifies with this approach semantic fields, which support the narration best. Afterwards the found terms have to be translated into a visual form language.

3.1.1 The Environment

A clear example for the narrative design is in our case the earth. It was important to capture the fascination of an orbital flight. Although the earth as a simple background object does not contribute to the actual game play, it is significant for the atmosphere of the overall game and makes the gaming experience authentic. It increases the depth of the game on a visual and emotional level.

Media consumers of our time are familiar with images from the orbit, like for example from the International Space Station (ISS) or the space shuttle flight, that it was clear that we have to keep close to these pictures in order to keep up with the plausibility of the game. An abstraction of the orbital area was therefore not acceptable.
It was important to find out what are the main impressions which are contributing to a realistic representation of the earth as it is perceived from space. Thus it was decided to integrate the following aspects:

- A high level of detail of the earth's surface including the continents as well as the oceans.
- A high level of detail of the lower cloud layers including the shadows which are cast on the surface.
- Realistic depth increase for the density of the atmosphere including the typical blue ozone coloring.

Shader models were developed which were used by the graphical effects developers, who had to create a real-time display of the environment.

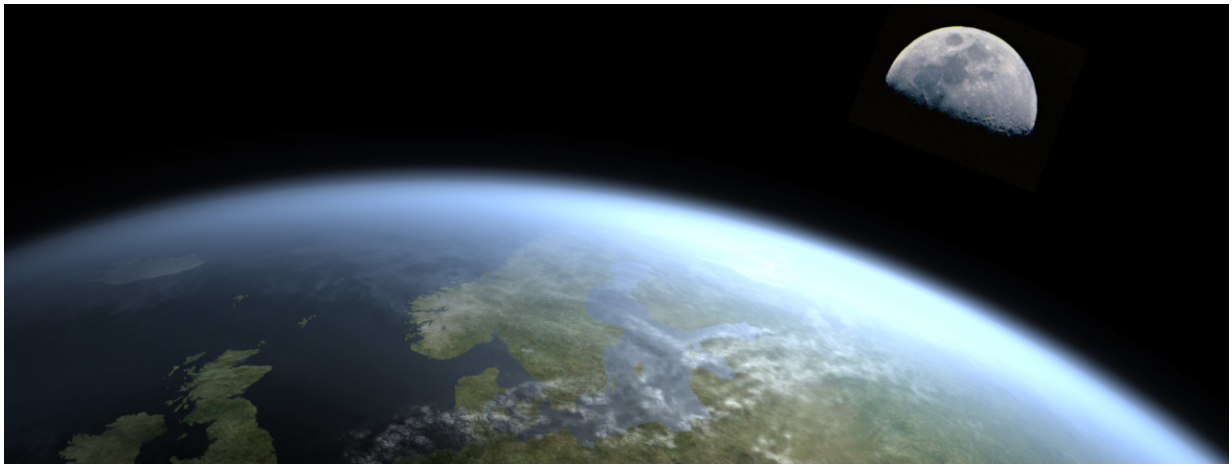

Figure 3.1: Earth's Orbit

Figure 3.1 shows an initial concept rendering of the earth's orbit illustrating the glow effect through the ozone coloring and the atmosphere layers over the continents.

3.1.2 The Trash

The trash should provide highest possible density to increase the 3D experience and support the depth perception using stereoscopic display. This meant the generation of a vast amount of objects.
An early rendering of the objects is presented in Figure 3.2. Satellites and trash pieces are symbolized in this illustration by cubes.

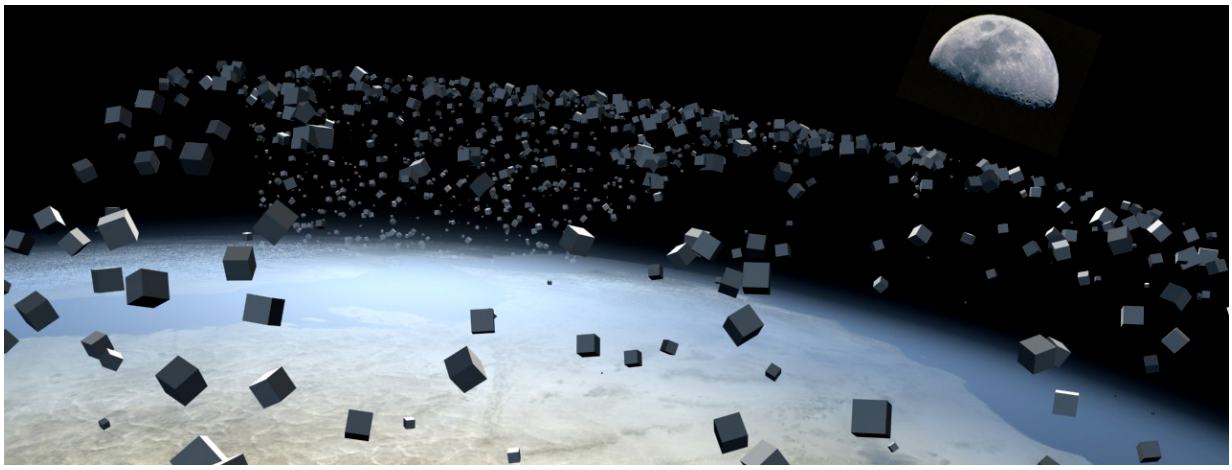

Figure 3.2: Trash Density

These objects had to be created very simple in order to reduce the amount of work time as well as the amount of polygons. Another aspect of the trash pieces was the possibility to separate them, in order to generate new trash if collision occurred with a trash piece.

Figure 3.3: Separatable Trash Pieces

As with the design of the earth visualization with these objects initial research was needed to start the design. We wanted to identify which materials were formerly used and how satellites were built. This was important since the trash of the old satellites and the newly started ones had to be distinguishable. A modular workflow was created, which started with the development of smaller portable elements that could be used as well as a basis for more complex objects. With this approach it was possible to generate a huge variety of trash pieces based on the individual components.

3.1.3 New Satellites

In the actual game new satellites are launched inside previously cleared orbital areas. Besides illustrating that they are new, they should express their type. Three different types had to be designed: scientific satellites, telecommunication satellites and military satellites. All satellites had to unfold during the launch out of a rocket capsule. A significant shape had to be designed and a hierarchy had to kept, that it was possible to unfold them with basic translation and rotation operations. Another challenge was to keep the satellites in the proportions that they would fit inside the rocket capsule.

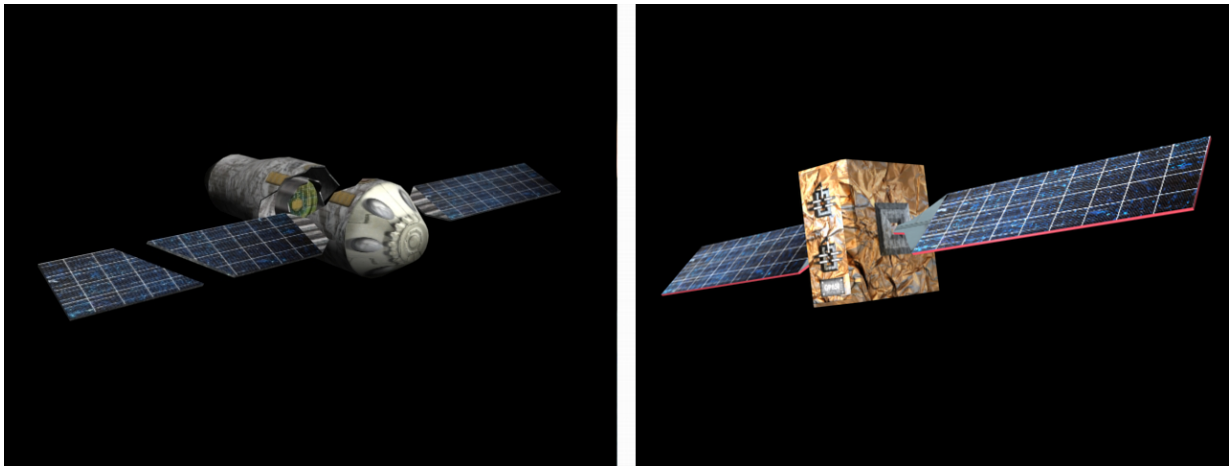

Figure 3.4: New Satellites

Figure 3.4 illustrates two new satellites. On the right side a fully functional satellite is shown, while the left side of the illustration shows a separated satellite.

3.1.4 Space Ships

In this design aspect it was necessary to distinguish the ships according to the story line and to characterize the players with their different backgrounds. In our case it was important to illustrate that the ships are made up from different trash and electronics pieces. It should be clear from the design who is controlling the hackers' ship and who is controlling the old mens' ship.
Both space ships had to have a plausible cleaning up mechanism that it was possible for the enemy team to perceive if and how trash pieces were collected. Thus it was important to create a reasonable space dimension that the collected geometries would fit in the bays of the ships.
They had to be both optically significant, where the hackers' ship was designed to, create associations with cartoon characters and that it was easy to get the impression that the typical nerd as an anime consumer would design a ship like that. The goal was to create a transformer-like design, which was not able to be transformed a misfortunate prototype which expresses the overly ambitious trash collecting team. Through LEDs and circuits the outer hull of the ship emphasized the programmers' background.

For the old astronauts' team a design was developed which is oriented on the early days of air and space travel. The ship is modeled as a kind of biplane, which through its plump shape expresses the age related slowness of the retirees. As the hackers' ship the shape can be easily perceived from distance and is unique.

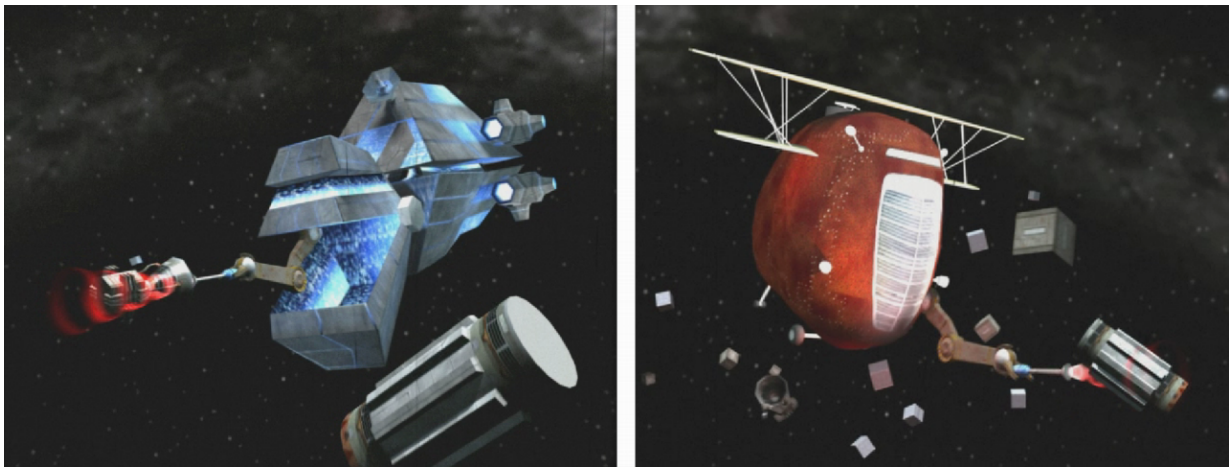

Figure 3.5: Space Ships

Figure 3.5 shows the ships of the two competing parties collecting space trash. The hackers' ship is illustrated on the left side, while the right side of the image displays the old mens' ship.

3.2 The User Guidance Design

These design elements were mainly focusing on the user support. They are characterized by their high information content and help the players to focus on the game play.

3.2.1 The Head-Up Display

In order to keep a consistent style of the information display an early concept of a Head-Up Display (HUD) was designed. An important aspect of the HUD was to keep its look as far as possible from the 3D elements in the game. Thus the graphical elements were created light emitting and with large clear shapes. Three different information units exist:

- The artificial horizon
  It provides information to the user on the position in the game area as well as information where trash pieces are located. The horizon is pitch sensitive and mapped on the ships orientation.

- The grabber orbit
  This object provides information to the player whether an object is in grabbing range. Furthermore it illustrates the score value of the trash piece, as well as the type of trash

or satellite. If the object is targeted the visualization is changed, displaying that the trash piece can be collected.

- The quadrant grid
  This grid illustrates the current game area in order to provide additional information where trash pieces have to be picked up. It is mapped directly into the three-dimensional space.

3.3 Modelling and Texturing of *Space Trash* Objects

Generally the *Space Trash* objects consisted of low-poly models, which were created with a common 3D software. By developing 3D models with a reduced amount of polygons and designing more detailed image textures at the same time, a three-dimensional impression was given during the game.
The modelling of a piece of trash, for example started with a basic shape of a cylinder which was cut into two pieces and was given a thickness of the surrounding edge. After this step the texturing of the object followed. In order to find interesting results, changes were bi-directionally made between the 3D software and the 2D image editing software. Therefore the 3D software was used to model simple shapes of for example: cable conduits, electronic devices, etc. which were rendered as an image file and transfered into the 2D imaging software in order to continue to work on it. This process is briefly illustrated in Figure 3.6.

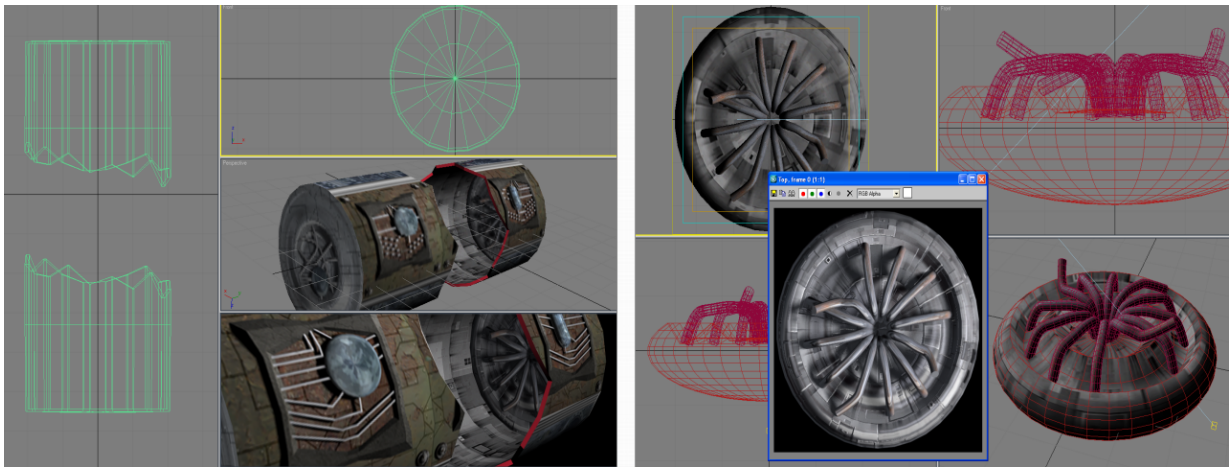

Figure 3.6: Low Poly Modelling

The task of the 2D image editing software was to make it possible to put different image layers on top of another, to vary between the shapes of the tip of the ball pen for blurring hard edges, create transitions, transparencies or just change the opacity.
The last step during this workflow was to adjust the texture elements to the unfolded 3D model and thereby prepare the files with an ”unwrap-uvw-modifier” for all subsequent coding of the game engine as illustrated in Figure 3.7.

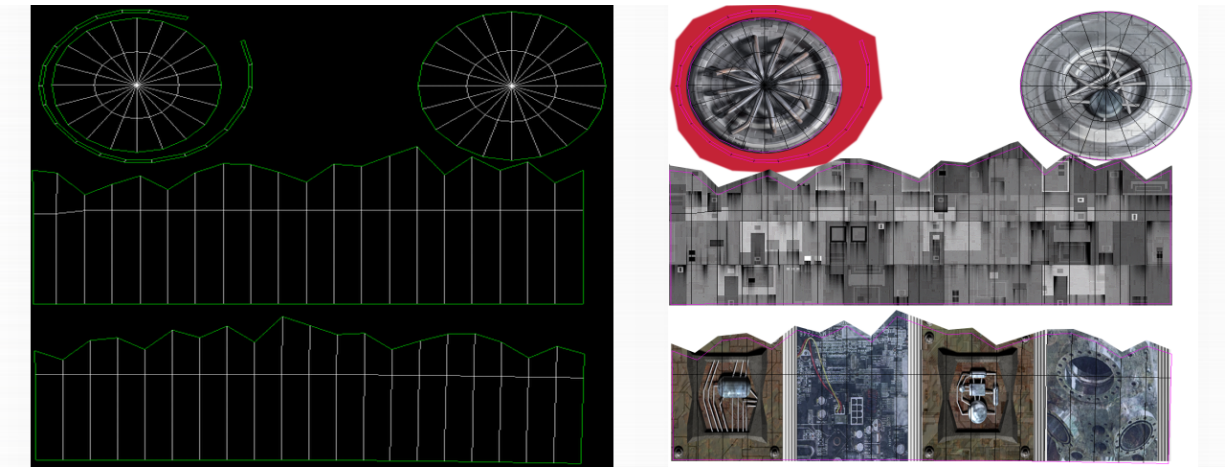

Figure 3.7: Unwraping Textures

3.4 Summary

This chapter has described the two main concepts of the visual design of *Space Trash*. The narrative design has emphasised the atmosphere and the setting of the game while the user guidance design has introduced the design element which were used to support the game play. A brief introduction on how the 3D models of *Space Trash* were created was given.

Chapter 4

Input Devices

> Any sort of ship you have to learn to pilot; it takes a long time, a new set of reflexes, a different and artificial way of thinking. Even riding a bicycle demands an acquired skill, very different from walking, whereas a spaceship - oh, brother! I won't live that long. Spaceships are for acrobats who are also mathematicians.
>
> Juan Rico - Starship Troopers

The physical interfaces are designed to encourage the intuitive user interaction and physical game experience. According to Frasca the game experience of the user is achieved through three main components: the rules, the fictional world, and the user interaction [Fra07]. By creating such a custom physical interface that enables us to design the user interaction scenario affects the experience of the whole game significantly.
The three main controlling tasks of the navigator (navigation with pitch and heading), the accelerator (speed control including back and forward), and the grabber (manipulation of arm with pitch and heading) are assigned separately each to an individual interface.
We used trash as well as recycled objects to create the interfaces as implication to the theme of the game. Additionally the use of well known objects gave preliminary information to the users on how the interface should be handled. An example for such an object is the interface made out of a bicycle wheel and pedal. It is obvious for the users to push the pedal with their feet as they do with bicycles.
The physical construction and aesthetics of the interfaces are not identical at the two sites, due to the use of previously used materials, and the design decision to create a different look and feel for each character background story of the site. In the following sections, we describe in detail the construction of the interfaces of a single site. The interaction sensing mechanisms and the basic interaction scenario are kept identical for both sites while the mounting techniques differ from site to site.

The sensor inputs from each interface are received via the analog ports of an Arduino board [ard] which is the converted into digital data and sent to an input server via USB serial communication. At the server it can be used for further distribution to the remote clients as described in Section 5.2.3.

4.1 The Navigator

The navigator controls the orientation of the space ship in pitch and heading. For the heading control, we employed a limitless turning wheel, as it is used with old sailing ships. The detected turning of the device is translated into a relative direction change rather than fixed angle such as implemented with car wheels.

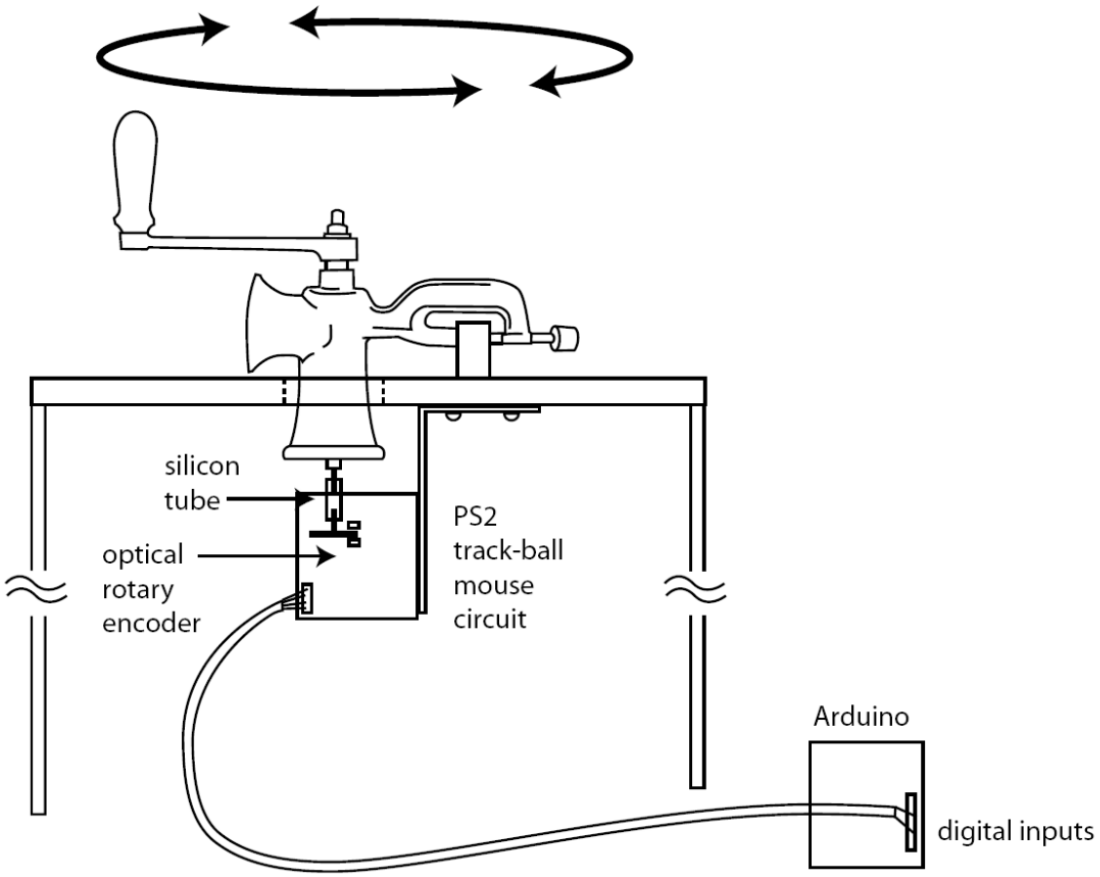

Figure 4.1: The Navigator

To enable this navigation method, an optical rotary encoder taken from a PS2 track ball mouse was used. For the physical turning mechanism, a modified old metal meat grinder's handle was incorporated. By controlling these existing objects a positive effect on providing a robust and familiar interface is given. I.e. the metal structure of the meat grinder is already designed for people to turn the handle with full strength; moreover people have preliminary knowledge on

how and in which direction to move the device when they see such a handle. The axis of the handle is connected with an optical rotary encoder. Using the PS2 mouse, allows us to receive the desired data through a PS2 connection. A PS2 library for Arduino [ps2] interprets the PS2 outputs.

4.2 The Accelerator

The accelerator controls the forward and backward movement of the spaceship. The back wheel of an old bicycle is modified and employed as an interface (see Figure 4.2). As a player turns the bicycle wheel with pedals, the motor attached to the wheel turns and generates voltage.

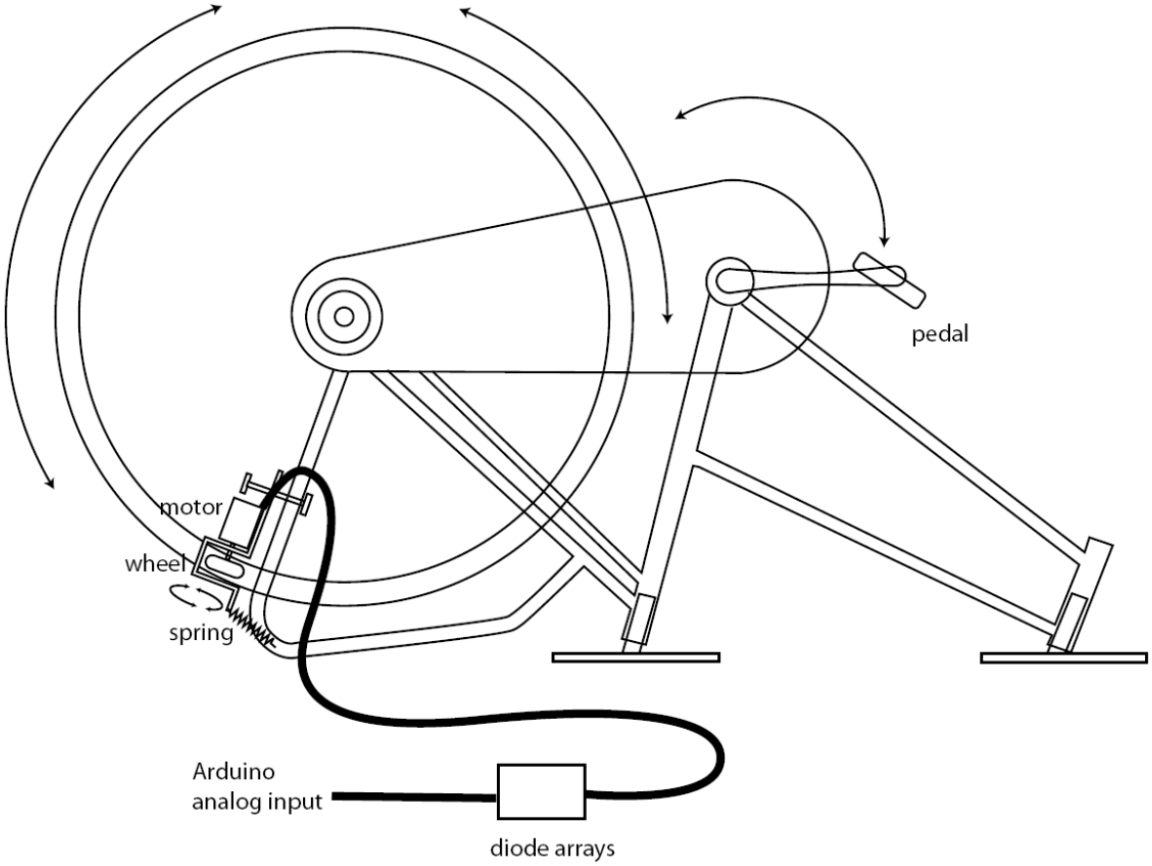

Figure 4.2: The Accelerator

We have used a DC motor with a low speed to generate voltage when turned. The motor acts as a sensor detecting the turning direction and the speed of the wheel. We employed a changeable resistor to lower the measured voltage in order to adjust the speed-signal ratio. Three arrays of diodes are added to limit the maximum signal from the sensor to about 2.3v in order to to protect the microcontroller. The microcontroller (ATmega168) uses an input-voltage between 0-5v on its analog input ports. The described motor sensor generates -2.3v to +2.3v. Therefore we add an

extra 2.5v to the generated signal to detect the negative voltage thus distinguishing the direction of the turn.

4.3 The Grabber

By moving the grabber interface, a player can aim the target point for the 3D graphical arm representation that collects the space trash.

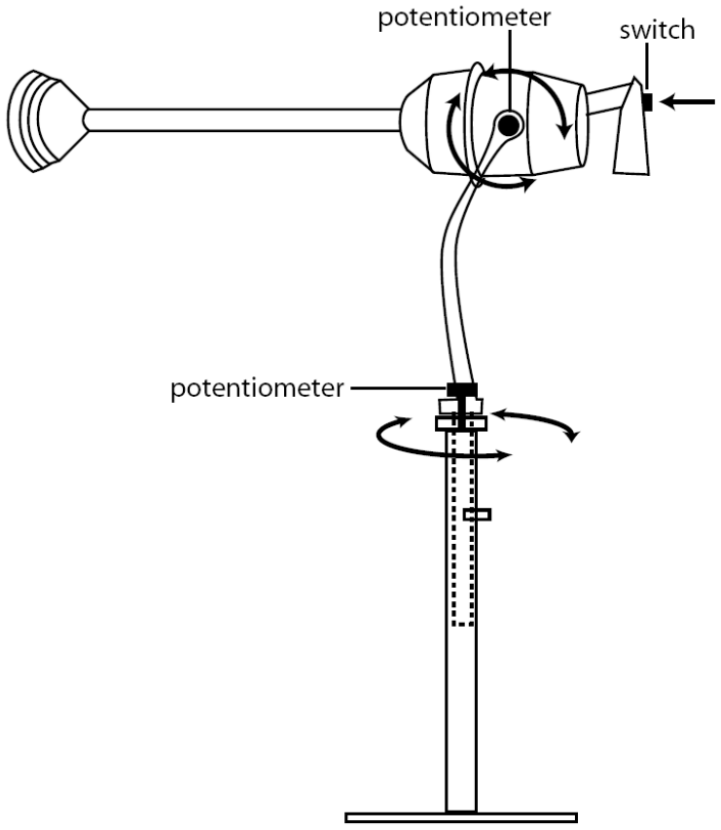

Figure 4.3: The Grabber

The physical interface of the grabber is created as an aiming arm mounted on an axis that is held on a metal base which rotates again on its own axis. This way the arm could move with 2 axes allowing the rotation of both the pitch and the heading direction.
The arm part is built out of an old washing machine. A linear potentiometer is attached to each axis in order to detect the turned angle. A small button or switch is added to the handle of the washing machine which can be pushed by the player if trash collecting is desired.
Figure 4.4 illustrates the devices being used at the European Researchers' Night 2008. The left part of the figure shows the navigator, the middle part the accelerator and the right hand side the grabber.

Figure 4.4: The Actual Devices used at the Researchers' Night

4.4 Summary

This chapter has provided an overview on the implementation of the different physical input devices. Technical sketches illustrated the mechanics, while the electronics used to implement the interfaces were briefly described.

Chapter 5

Software Architecture

That's no moon, it's a space station!

Obi–Wan Kenobi - Star Wars

This chapter will give an introduction into the software architecture of *Space Trash*. The foundation for the application is the *inVRs* framework, which was designed to create immersive networked virtual environments.
The first half of the chapter will provide a rough overview of the *inVRs* framework, while the second half will concentrate on the specific *Space Trash* architecture.

5.1 *inVRs* Architecture Overview

inVRs consists of input and output interfaces for the interconnection to different scene graphs and the access of a variety of input devices. In *Space Trash* this functionality is used to support newly developed physical interfaces. Three independent modules support interaction, navigation and network communication. The modules and the interfaces are connected to a system core, which manages communication between the components using discrete events and continuous flows of transformation data packets. Inside the system core a data storage system keeps the logical entities of the NVE as well as data about the users interconnected with each other.
In general three main types of components exist. Figure 5.1 illustrates the main components of the *inVRs* framework. The interfaces for input and output are shown in grey, the modules are drawn in light blue while the system core with its subcomponents is displayed in a darker blue.
Additionally many smaller features like logging functionality, math functions, and data types are integrated in the system core. More detail on the overall architecture has been previously published in [AV06].

5.1.1 System Core

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

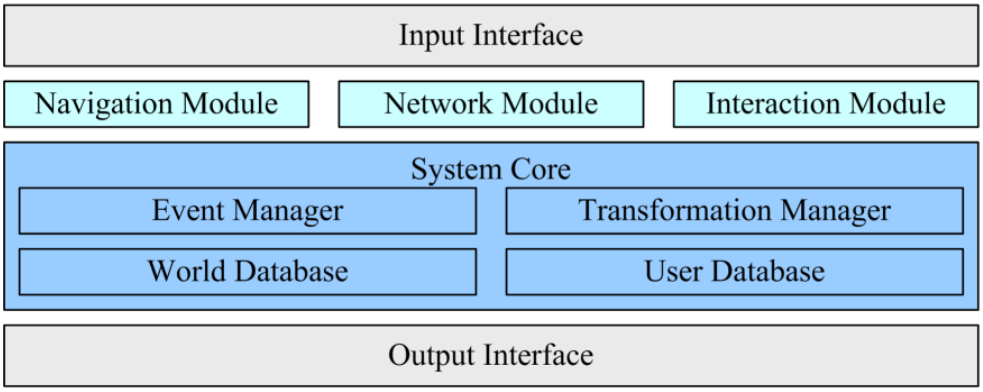

Figure 5.1: The Basic *inVRs* Components

database and the user database.

Databases

The world database is responsible for keeping the layout of the VE including the description of its components. It acts as a high-level manager for the geometrical transformations of the VE objects. Several types of objects are available in an *inVRs* virtual world.
So-called environments do not have a graphical representation. They are coordinate systems that are used for grouping and thus the support of culling sub-objects. These environments are often used in conjunction with the network modules to split the NVE into several sub VEs.
These sub-objects could be either tiles or entities. Tiles are always fixed they can be used as decorative parts of the VE representing buildings, parts of a landscape or other static objects. Tiling mechanisms can also be used in the framework to split large datasets into disjoint parts.
The most interesting objects in the world database are the entities, which are typically used for interaction. To develop complex virtual worlds it is common to define own entity types and equip them with application specific functionality.
In Figure 5.2 the scene graph representation of environments, tiles and entities is illustrated.
The second database - the user database - manages the local and the remote users of the VE. It keeps the coordinate systems of the user representations including the cursor data and links these transformations onto the graphical representations stored inside world database of the system core.

Communication

The communication architecture of the *inVRs* framework differs significantly from other solutions in the field. The framework makes a clear distinction between two types of data - events and transformation data packets.
Transformation data packets contain geometrical transformations, which can be applied on objects of the VE like entities or the camera. The transformation manager handles these packets. It is not only used for the distribution of the data, but rather performs significant modification of

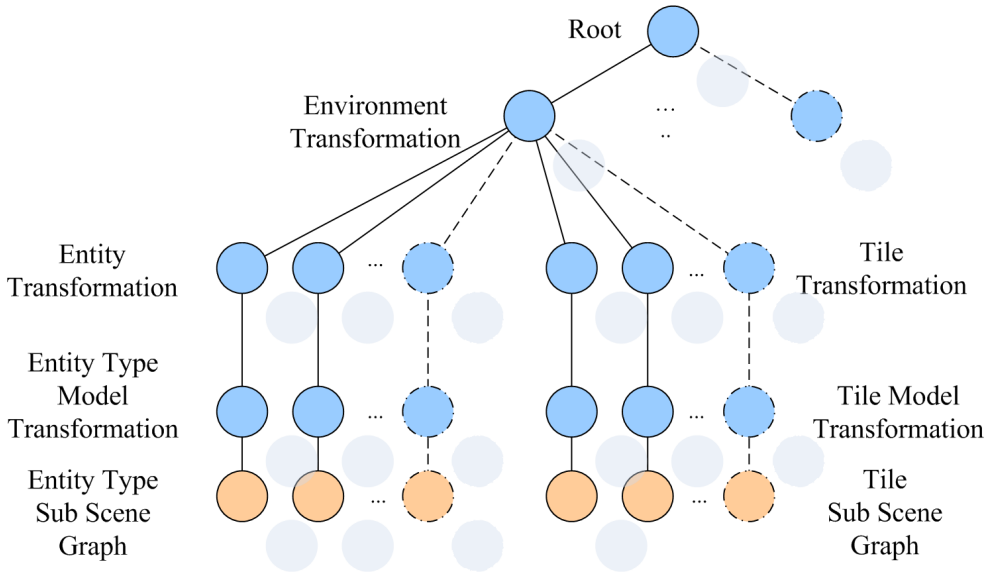

Figure 5.2: The Transformation Hierarchy of the World Database

the transformation by piping the packets through different stages of the modification process. The events are discrete messages. Events are to be distributed in order from component to component. Event cascades where one event triggers another are to be avoided by the application designer.
In case a network module is available transformation data is typically transmitted via UDP in an unreliable manner and events are transmitted via TCP in a reliable way. The concepts of transformation management and the event system have been previously published in [ALBV07].

5.1.2 Interfaces

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

Input Interface

A mapping from the data gathered by the input devices is performed on an abstract controller, which can be accessed from the modules or the application. The input data is processed, abstracted and exposed in the form of buttons, axes and sensors.
The *inVRs* input interface plays a major role for the development of Space Trash, since the physical interfaces which were previously introduced had to be interconnected to the application.

Output Interface

The current implementation of *inVRs* provides an abstraction layer for scene graphs and audio output. By using the OpenSG multi-display capabilities is easily possible to render *inVRs* applications on CAVE-like [CNSD+92] devices, curved installations like the i-Cone [SG02] or Head-Mounted Displays (HMDs) [Sut68] as well as simple monoscopic desktop systems.
For audio output currently only OpenAL is supported. The basic functionality of playing and stopping audio files is provided so far.

5.1.3 Modules

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

Navigation

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

Interaction

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

Network

The network module is implemented using a two-layered approach. The top layer the high-level interface provides common access to all *inVRs* and application specific components. User defined messages, events and transformation data packets can be distributed to all other participants or a defined Area of Interest (AOI).
The low-level component of the module is designed to be exchanged and to implement specific

network protocols. The communication topology and the database distribution topology is fully implemented in the low-level component and hidden from the application developer. Additional optimizations like AOI management are handled as well by the low-level component.

5.2 *Space Trash* Architecture

This section describes the architecture of the *Space Trash* application as well as internal mechanics of the implementation. It explains how the various aspects of the game are reflected in the source and how these components interact with each other.

5.2.1 Gameplay Perspective

A game is usually built up from a number of elements, called entities. An entity represents an object in the world that acts within an environment. In our application these would be:

- the player ship(s)
- the game field
- a sub area of the game field which has to be cleaned
- thrash items
- satellite items

The players ship is the entity that the player interacts with in an immediate way. A player ship acts as an avatar. As the *Space Trash* application is controlled by three players, the ship acts as the common point of identification.

5.2.2 Physics Perspective

For simulating most of the game logic, *Space Trash* uses the *inVRs* physics module which is built upon the Open Dynamics Engine (ODE). Every type of entity has a fixed collision bounding volume. The fixed physics bounding is instantiated for every actual replication of the entity using the entity's world transformation as its offset. As the physics calculations are relevant to the game logic only the server evaluates the results for the physics which are then replicated across the network to the client. Some entities, namely satellites as well as trash parts, can break into smaller chunks of trash. More detail detail on the physics module is available in Section 6.4.

5.2.3 Network Perspective

The implementation for the game is divided into two main components, server and client, as well as an input server. The server is responsible for running the game logic and distributing the results of the interactions to all other participating clients. The advantage of using a server instead of a peer-to-peer (p2p) implementation (which is one of the standard implementations of the *inVRs* network layer) is that there is always a single point in the network where game play decisions are made. Using a p2p approach would complicate the issue as these game play decisions would have to be synchronized and possibly even voted on in order to decide which result is the correct one.
Since the server runs the game logic (and therefore the physics as well) there is little to do for the client side. The client is merely a representation layer to the player. It receives the input and sends it to the server which corrects the game state accordingly and the results are send to back to the client in order to be displayed to the user. Since the round trip to the server is subject to latency the values sent across the network are extrapolated in order to smooth the transformation. The logic for this transformation handling is implemented using the *inVRs* transformation manager which extrapolates matrices inside its pipes.

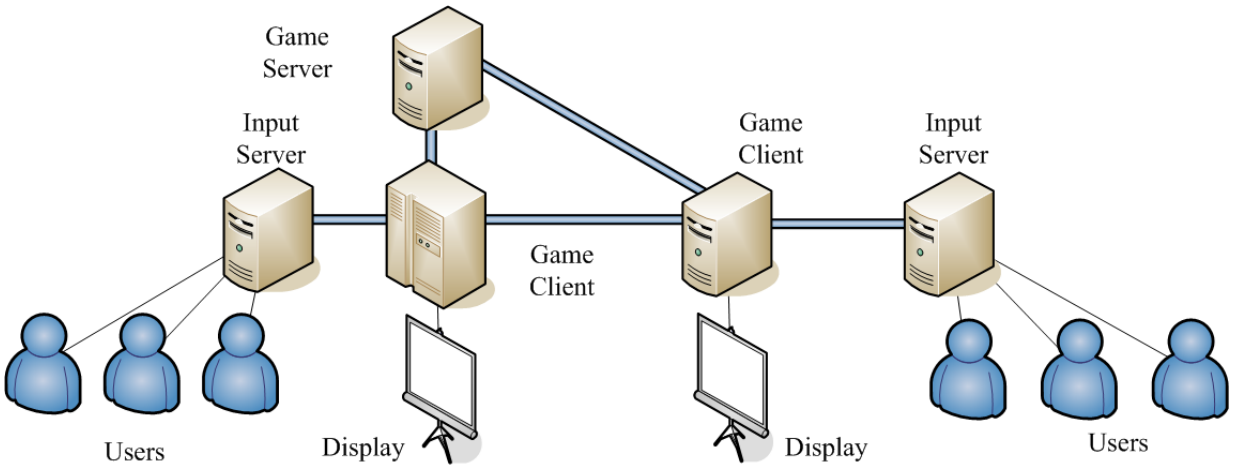

Figure 5.3: The Game Topology

Figure 5.3 illustrates the network setup for the game with two teams. The communication between the `InputServer` and the `GameClient` is done via the User Datagram Protocol (UDP). Information between `GameServer` and `GameClient` is sent via either UDP or the Transmission Control Protocol (TCP) depending on the importance of the event: Gameplay relevant information is sent via the reliable TCP channel, whereas transformation updates are transported via UDP.

5.2.4 Source Code Perspective

From a programmers point of view, the source of the *Space Trash* application is divided into multiple source directories which makes task distribution easier in a team. Table 5.1 gives a

short description of the source code folders and their content:

Module	Description	Belonging
animation	code relevant to the animation module	client only
camera	code for following and loading camera paths from 3ds files.	client only
client	code specific to the client binary.	client only
collada	collada xml parser	client only
common	common code between client and server	client and server
events	*inVRs* events sent between client and server	client and server
HUD	Heads-up Display code	client only
ik	inverse kinematics code for grabbing arm	server only
inputserver	inputserver forwards input data to client binary	stand alone
server	server specific code	server only
testEntities	entity viewer (for adjusting physics properties)	stand alone

Table 5.1: Source code folders, description and to which binaries they belong

5.3 Implementation

The *Space Trash* application is developed using the C++ programming language. The implementation aims to employ modern libraries such as boost wherever feasible. Most of the development was done using Eclipse as an Integrated Development Environment (IDE), with Subversion (svn) as a version control system and trac as a web based team support and ticketing mechanism. The source code used to be compiled using a classic makefile build system but has recently been ported to the more contemporary cmake build system, which provides better flexibility. The development process was split into several smaller teams consisting of one to three people each responsible for one or more of the core components. Most of the development was done on a single branch, relying on each other and making people feel bad for breaking the head revision.
As the final target environment was multi-cpu itanium system, an SGI PRISM, whereas the development was done on standard x86 desktop hardware, regular tests on the target environment had to performed. Due to the distributed development process part of the initial design was to introduce common coding conventions and a initial code structure.
As the application is separated into three binaries, GameClient, GameServer and InputServer, the code was developed as two distinct source trees. In order to avoid code duplication and because of required shared interface between those two applications, a common code folder was introduced. It contains some simple helper-classes and enumeration data types. The code of the input server was kept together with the client code.

5.3.1 Client

Comparing *Space Trash* with the classical Model View Controller (MVC) approach, the client is considered as a view as well as a controller to the common game state. It does not deal with any

game related decisions, but rather acts as a simple proxy for user input and a visual display for the game. As *inVRs* takes care of most of the communication and basic infrastructure, such as dealing with virtual reality display and input hardware, the client can be considered a thin client. The following important components are included in the client binary:

- HUD
- a simple wrapper object for the earth visualization
- audio output and background music helper classes
- user navigation
- user interaction
- satellite launch animation
- camera flyby and credits

Game Loop

In parallel to the global, server managed game state, it is necessary to maintain a local state machine (see Figure 5.4) on the client in order to differentiate between waiting states, the gameplay itself, the results screen, and the credits. Furthermore some internal initialization and cleanup is handled through this state machine.
The client-side game loop consists of the following states:

FIRST_DISPLAY After the *inVRs* framework has been initialized, application specific setup tasks have to be performed, such as joining a team and triggering the HUD setup.

WAITING_FOR_READY In order to start a new round, a minimum amount of teams have to be present and acknowledge their being ready for the next iteration of the game. As soon as all teams have pressed the trigger button, indicating this being ready, the state machine continues with at the COUNTDOWN state.

COUNTDOWN This state is a timed, automatic transition state, which displays a visual countdown timer to the players.

GAME_RUNNING This state is active during the actual interactive gameplay. It updates the HUD and sound classes and checks for changes resulting in a termination of the game.

SHOW_RESULT After one round of the game, the players are informed of being the winner or having lost the game to the other players. A more sophisticated results table is currently under consideration.

FLYBY As a lot of man power and good will has gone into the creation of *Space Trash*, a credits sequence with a camera fly-by is shown.

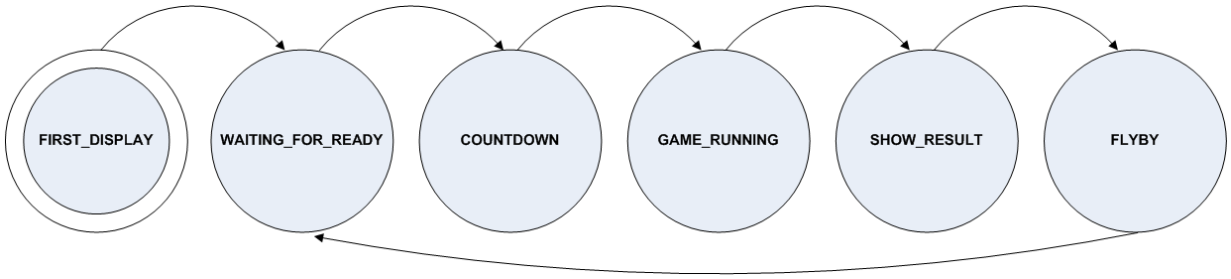

Figure 5.4: The Client's Game State Machine

Entity Meta Information

As the HUD and user manipulation classes require a mechanism to determine various meta information of the gameplay relevant entities, a EntityInfo class singleton has been created. It builds a table of this information on the fly with information sent from the server. It offers a basic, pragmatic interface to query for different properties of known entities.

Input Server

The third part of the system is the input server. It is responsible for reading input values from the arduino board and sends the information via UDP to the input system.

5.3.2 Server

The server is the central point within the application logic. It receives input from the clients, updates their positions, performs all physics calculations and distributes the results back to the clients. It is also the instance in the network that performs the gameplay decisions and keeps track of the scores. From an *inVRs* point of view the server is one peer in the game and only our implementation of the communication protocol the server node is distinguished as a logical master.
A basic Unified Modeling Language (UML) overview of the major game server components and their relations to *inVRs* provided classes is illustrated in Figure 5.5.

Mapping between *inVRs* and GameLogic Entities

A common approach in *inVRs* is to derive from the *inVRs* entity class to define new entities and their functionality. Modern object oriented paradigms propose to use composition over inheritance which we decided to embrace. We therefore have a hierarchy of GameEntities which manage and relate to *inVRs* entities.

Areas

The game field is split up into multiple target areas. As with the inVRs-entity-to-GameEntity relationship we were able to directly map *inVRs*' Extensible Markup Language (XML) defined

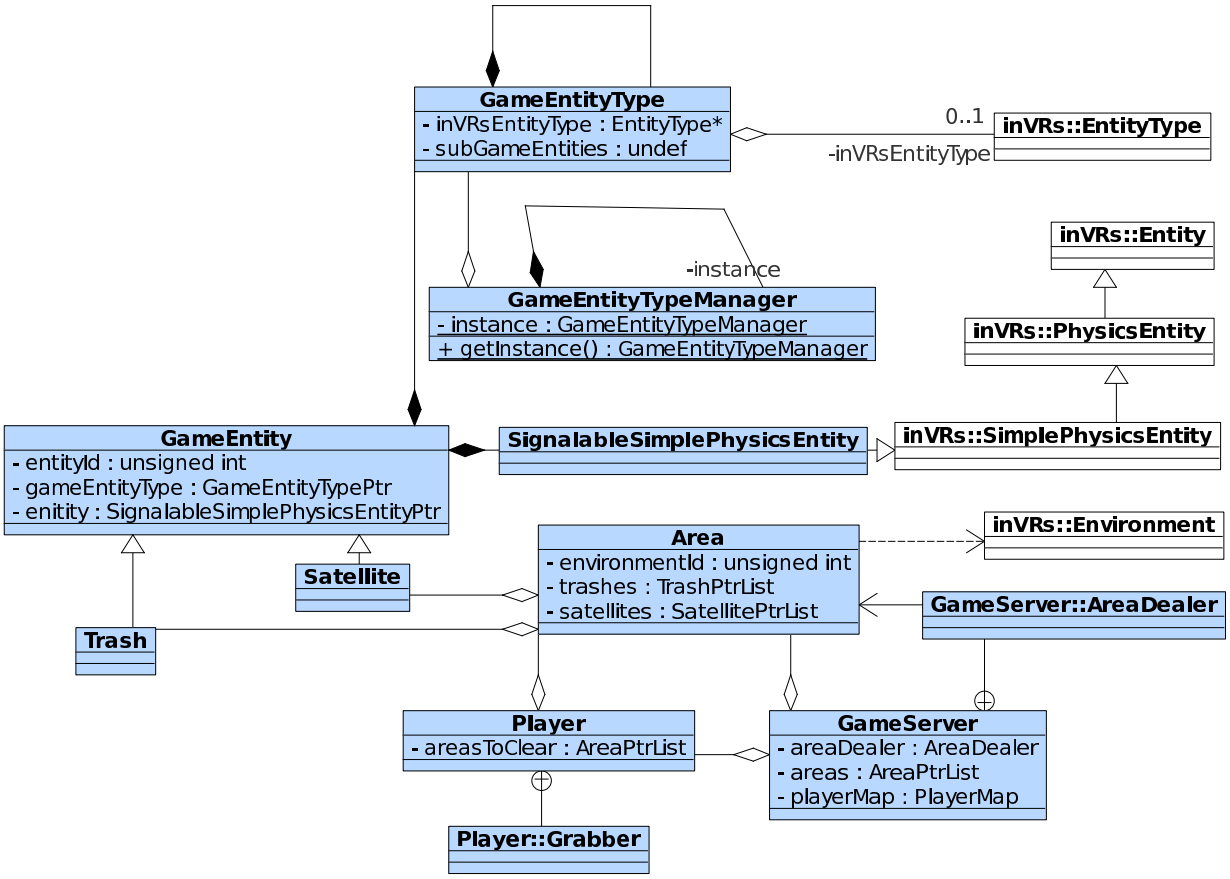

Figure 5.5: Gamelogic Classes

environments to our area class and concept. An area may contain multiple game entities, whose position is dynamically updated through the physic simulation. When an entity leaves the physical boundaries of its current environment, *inVRs* internally handles this movement into another environment. An area registers a callback with the environment in order to be notified of such a transition and reflect the area change within the game logic.
As areas have to be cleared by a team, every team has one or more assigned target areas. The area dealer is responsible for assigning areas to teams. There are two different distribution modes a random and a first-in-first-out (FIFO) mode. The random mode generates a random number to select the next target area for the team, while the FIFO mode uses a predefined list which allows a more specific definition of the game flow. This mode is especially useful for live presentations with first-time players.

GameEntites – Trash and Satellites

Within the *Space Trash* application there are two major game entity types, trash and satellite. There are two marker classes that derive from GameEntity, but most of its logic is shared and contained within GameEntity. A GameEntity has a specific GameEntityType which contains

information about how the entity is built up, whether it is destructible and, if destructible, which sub components the entity would break into. The GameEntityType also contains game play relevant information such as a score, a display name as well as physical properties required to break the object. Once again every GameEntityType has a corresponding *inVRs* EntityType following the composition-over-inheritance principle allowing us to reuse the existing *inVRs* components without complicating the inheritance chain due to framework design considerations.

GameEntityType Manager

The GameEntityType Manager is responsible for parsing a XML file that describes the GameEntityType and their properties as well as their physical structures and the parts they would fall apart to if a collision with enough force happens. The structure of the XML file is coherent with *inVRs* entity type declaration files and allows us to tag game entities with additional information. Another responsibility of the GameEntityTypeManager is to maintain a registry of known types and allow look-ups by *inVRs* entity id as well as a look-up by name.

MainLoop and Global Game State

Initially the GameServer's state machine (see Figure 5.6) is in the EMPTY state, waiting for players to connect. For every client that registers with the server, the server instantiates a Player object that manages the teams' ship entity, score and a possible list of assigned target areas to clear. If enough clients (configurable via XML) have connected, it initializes all clients and assigns them their ship entities. After initialization of the clients, the server waits for all teams to indicate being ready after which the server passes through an internal PREPARE_ROUND state which sets up all internal entities and prepares the game field.
After initialization a countdown timer is triggered and the round starts after this defined period of time (also configurable via XML) and the state machine enters the GAME_RUNNING state. In this state the server now updates the physical simulation, handles game related events and checks whether the round has finished. This criterion is met if either the round time has ended, all target areas have been cleared or too many players have left the game and the minimum number of players is no longer reached.
If the end of the game is reached, the server switches to the GAME_END state and cleans up internal data structures, such as the areas and entities, which happens while the client is showing the results and credits. After successful cleanup the state machine enters the initial empty state again.

Messaging

Our Events a categorized into two major sections, client-to-server and server-to-client events. Since *inVRs* provides the necessary infrastructure to synchronize object transformation among peers, we were left with having to implement the game logic events only.
As the code for events is shared between the server and client code base, the client as well as

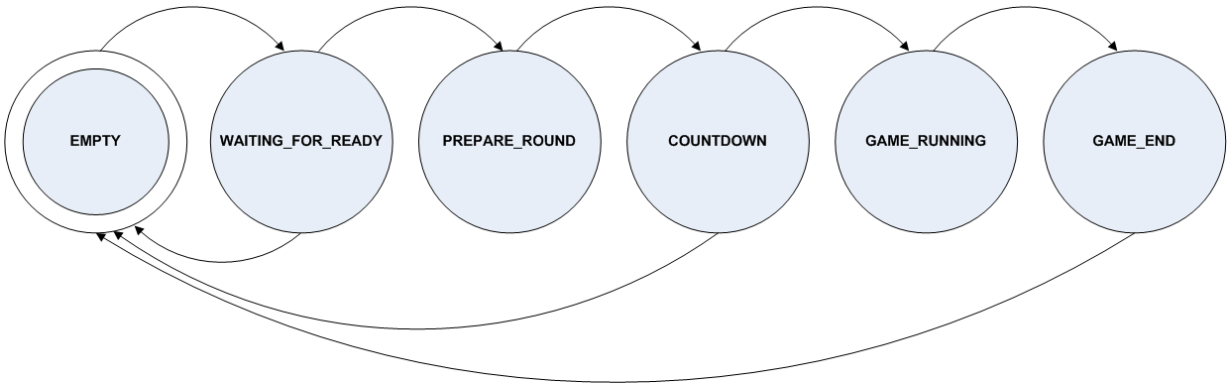

Figure 5.6: The Server's Global Game State Machine

the server provide and implements interface classes, GameServerInterface and GameClientInterface, in order to avoid direct, cyclic source code dependencies between the GameServer and the GameClient binaries. These interfaces consist of all the methods required by the events in order to execute.
The following events are sent from the client to the server and are derived from the abstract ClientServerEvent:

JoinGameEvent is sent to the server as soon as a player wants to join the game. It carries a parameter indicating the preferred team.

PlayerReadyEvent indicates that a player has pressed the ready button and agrees to start the next round.

Begin/EndManipulationEvent are used to indicate that a player grabbed a specific entity and prevents other clients from interacting with the same entity.

Other events, sent from the server to the client are derived from the abstract ServerClientEvent class and can be one of the following:

ShipEntityIdAssignmentEvent informs the team of the *inVRs* entity id of its assigned ship after it has been instantiated by the server.

PlayerScoresEvent contains information about all player scores.

Create/Update/DestroyTargetIndicatorEvent transports meta information about target entities from the server to the client for use within the HUD.

SetHighlightedEnvironmentsEvent propagates the assigned areas from the server to the client indicating to the user via the HUD which areas are to be cleaned next.

StartCountdownEvent is sent to all clients at the same time to ensure that all players are synchronously preparing for the round and to compensate for possible varying resource load times prior to actual start of the game.

Start/EndRoundEvent is sent to the client at the start and the end of rounds, indicating a transition in the GameServer's global state machine.

SoundEffectEvent triggers client-side sound effects on physical impact between entities.

Example Application Flow In order to explain in further detail how the message flow between the clients and the server looks like, we provide an example message sequence diagram between two clients and a server. (see Figure 5.7)
The first three conditions deal with the round setup and player initialization, the next two (hasScoreChanged, isAssignedAreaCleared) may happen during the gameplay, depending on the users actions. The last condition checks wether a round has finished.

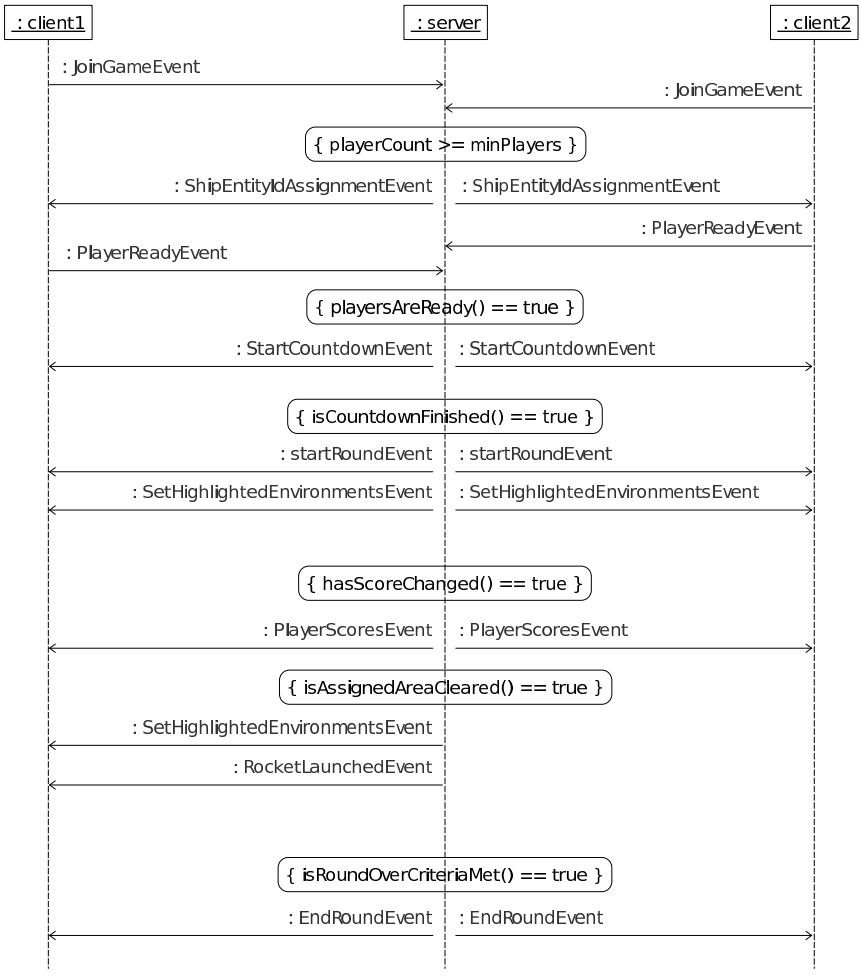

Figure 5.7: Example Event/Message Flow

Collision Detection and Response

The gameplay for *Space Trash* relies heavily on the physic capabilities of *inVRs*, which currently uses ODE via its useful C++ wrapper oops. *inVRs* further provides an `EntityType`, `SimplePhysicsEntity` which allows to easily extend existing *inVRs* entity type configuration with properties for the physics simulation. Once again this design allows the user to derive from `SimplePhysicsEntity`. We use this capability to derive a class named `SignalableSimplePhysicsEntity` which allows registering callbacks with this entity. This mechanism is implemented using boost signals.
On creation of a `GameEntity` a corresponding `SignalableSimplePhysicsEntity` is instantiated and callbacks with this *inVRs* entity. The `GameEntity` is then notified whenever the physics entity changes its environment or enters/leaves the physical bounds of the game field via the registered Signals. This callback is required to keep the information about which `GameEntity` is in which area in sync with the *inVRs* information about which physics entity is within which environment.

5.4 Summary

This chapter has given an introduction into the *inVRs* framework as well as the architecture *Space Trash*. The implementation of *Space Trash* is one of the bigger use cases demonstrating the flexibility and feature richness of *inVRs*. The development has indicated which improvements could be made to *inVRs* to further ease the rapid development of VR applications.

Chapter 6

Gameplay Components

> Mommy, how come all the other kids in class get new mops and I don't?
>
> Roger Wilco - Space Quest

The application consists of a large variety of components which had to be newly developed or were already available and had to be configured and enhanced.
In order to allow for user participation in the application navigation and interaction models had to be created. Especially the support of physics simulation of trash pieces and navigation is challenging with networked application.
In order to implement the user guidance design in the application a sophisticated HUD was developed.
This chapter will introduce the concepts of all these different aspects and provide a deep insight in the application mechanisms.

6.1 Navigation

This section describes how the data derived from the input devices, which will be introduced in Section 4, is used to move the user's ship through the scene. In addition to only moving the ship, physics simulation is incoorperated to simulate inertia of the ship. Furthermore the ship's roll while changing it's orientation and the it's tilt while moving up or down is realized by changing the orientation of the camera attached to the ship.
For forwarding the data to the application an input server was developed. It polls the data from the Arduino board (see Section 4) and sends it via UDP to the application.
As the *Space Trash* application is built using the *inVRs* framework, the navigation concept of *inVRs* shall briefly be described in the following. *inVRs*' *Input Interface* is used to abstract data generated by physical input devices. Every logical input device may offer a combination of buttons, axes and sensors. Several logic input devices can be combined to an abstract controller. The data from this controller is polled by three different navigation models implemented in *inVRs*.

These are models for speed, orientation and translation. Please see [AV06] and [AHKV04] for further information.

6.1.1 Implementation

Before discussing the implementation of the navigation itself, the developed logical input device has to be described. This device is not only used for navigation but also for interaction.

- *StrangeDevice*
  This class is part of the *Input Interface* and used to abstract the input of the physical devices. It receives the data from the input server and therefore combines all three input devices. The abstract device consists of five axes, one button and two sensors – although the real devices do not contain any sensor. The reason for that is, that no head and hand tracking is used in *Space Trash*. Thus the sensor values for head and hand are simulated. The head's tracking data is fixed while the hand's data is calculated using the axis values from the grabber. See Section 4.3 for a description of the grabber. The sensor value of the user's hand is necessary for interaction which will be described in the next section.
  The number of axes is put together by two axes from the Grabber, one from the Accelerator (described in Section 4.2) and two from the Navigator (see Section 4.1). The single button is located at the Grabber.

For navigation only two out of three defined *inVRs* models are used. The models for speed and translation are combined into one model. That is because for physics simulation an impulse, composed of speed and direction, is needed (see Section 6.4). As orientation model a pre-implemented *inVRs* class is adopted.

- *AllInOneTranslationModel*
  This class combines the models for speed and translation. It gathers the data from the abstract controller in order to change the ship's transformation in the virtual world. In detail the data from the accelerator and form one axis of the navigator is used. The speed and movement direction of the accelerator's wheel determines how fast the ship moves forwards or backwards, while up and down movement is calculated using the axis value of the navigator. An additional pitch while moving up or down was simulated using the *TiltableCameraTranformationWriter*.
  This approach is adapted for running *Space Trash* on a desktop system by the class *AllInOneTranslationButtonModel*. Instead of axes values from *StrangeDevice* button values (e.g. from the keyboard) are used to calculate the speed and translation of the ship.

- *OrientationAbsoluteSingleAxisModel*
  That is a class adopted from the *inVRs* framework. It calculates the ship's orientation using the data derived from the Navigator. The *TiltableCameraTransformationWriter* is responsible for simulating the roll of the ship while changing orientation.

- *TiltableCameraTransformationWriter*
  This class is used to alter the orientation of the camera depending on the angular velocity of the ship changing it's orientation and the speed of the ship moving up or down. This modification generates the impression of tilting the ship.
  For further informations about *inVRs*' transformation management see [ALBV07].

6.2 Interaction

This section explains how collecting trash in *Space Trash* works and how it is implemented. Furthermore it gives an overview over two different interaction techniques. One of them is used in *Space Trash*. The other one was implemented at an early development state and can be employed if other input devices are used.
The *Space Trash* application was intended to be displayed on a curved screen or a power wall. A user standing in front of this screen has to be able to interact with objects inside the scene. Simply mapping the real-world position of the user's hand into the virtual environment would significantly limit the range of interaction. Objects farther away than the user's arm length would not be reachable. Therefore there was the need for an alternative approach on interaction that enables the user to reach into the scene without directly moving into it.
Because at the beginning of the game design there were two options of usable input devices – a wand on one hand and a static input device on the other – two different interaction techniques were implemented.
In case of using the wand an approach of widening the user's interaction range is virtually changing the user's arm length. This method is called *Go-Go* [PBWI96] technique (see Section 6.2.1). The deployed input device is fixed at a certain position and only it's orientation can be changed. In order to enable the user to grab objects without changing the position of the physical input device, the **H**and-centered **O**bject **M**anipulation **E**xtending **R**ay-casting (*HOMER*) technique [BH97], as described in Section 6.2.2, is used.

6.2.1 The Go-Go Technique

The *Go-Go* technique [PBWI96] uses a non-linear mapping between the movement of the user's hand in the real world and the movement of the cursor in the virtual environment. As shown in Figure 6.1 $\vec{R_r}$ is the vector between the user's chest and hand. The vector $\vec{R_v}$ is the corresponding vector between the chest and the position of the cursor in the virtual world. R_r and R_v are defined as the lengths of those vectors. If R_r exceeds a defined threshold D, the length of the virtual arm R_v grows quadratically.
The mapping function F which is used to calculate the length of $\vec{R_v}$ has the following form:

$$R_v = F(R_r) = \begin{cases} R_r & \text{if } R_r < D \\ R_r + k(R_r - D)^2 & \text{otherwise} \end{cases} \tag{6.1}$$

where k is a constant value with $0 < k < 1$.

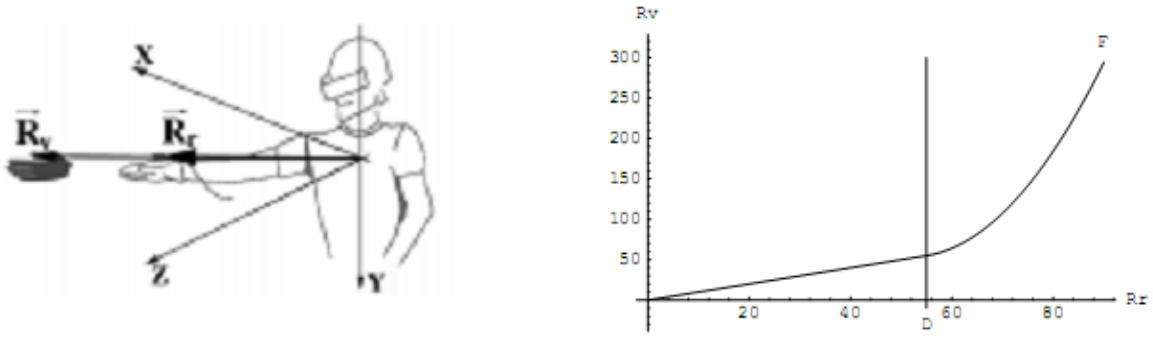

Figure 6.1: Calculation of the Virtual Arm Length [PBWI96]

If the user interacts with objects close by, a linear mapping between the hand and it's virtual representation is performed. As soon as the distance between hand and chest exceeds D the "virtual arm" grows and the user can interact with objects that would be out of reach if only linear mapping was used.
This approach was implemented as part of the *inVRs* framework but not used in the original setup of the *Space Trash* application because the input device for the cursor movement was designed in a way, that only the cursor's orientation and not it's position can be changed. But if one wants to replace the original input device with another, such as a wand, this implementation of the *Go-Go* technique can be used as interaction model for the *Space Trash* application.

Implementation

The *Go-Go* technique is implemented by a single class called *GoGoCursorModel*. This class calculates the transformation of the cursor based on the data collected by the input devices using Equation 6.2.1. Because only head and hand tracking is available, the position of the user's chest has to be estimated. It is determined by a configurable value representing the user's height of head. This value is subtracted from the z-coordinate of the tracked position of the head. The values for k and D can also be user defined. The configurations are set in an *inVRs* specific configuration file.

6.2.2 The HOMER Technique

The HOMER technique [BH97] is another way of extending the user's arm length. It is a combination of two interaction approaches: on one hand ray-casting for selection and a hand-centered manipulation on the other hand.
The ray-casting technique uses a ray that reaches into the scene as an extension of the user's arm. If the ray intersects with an object in the scene the object can be grabbed and becomes attached to the end of the ray.
As manipulating objects that are directly mapped on the user's hand transformation instead of a ray is much more intuitive, but ray-casting enables the user to select objects further away, the HOMER technique uses the advantages of both methods. A ray extends the user's arm and is

used for object selection. In case of grabbing the object, the cursor moves to its position instead of attaching it to the ray. This enables the user to interact with, rotate and manipulate objects in an intuitive way. After dropping the object, the cursor moves back to the position of the user's hand.

Implementation

The logical input device implemented for *Space Trash* which is used to abstract the physical input devices and to simulate the tracking data of the user's hand is called *StrangeDevice* and described in Section 6.1.1.
The grabber as a physical interface (see Section 4.3) is used to aim at entities that shall be picked. An invisible ray of limited length extending the grabber reaches into the scene. If the ray intersects with the bounding box of an entity the interaction state of *inVRs*' interaction state model, as illustrated in Figure 6.2, changes from idle S_0 to selected S_1. By pressing the grabber's button the selection is confirmed and the grabbing arm (see Section 7.1) moves to the bounding box of the selected object.
In order to collect space trash there is no need to give the user the possibility to manipulate grabbed objects. The cursor must move to the trash part, grab it and instantly move backwards with the object attached to the grabbing arm. This lead to some difficulties with the interaction state model implemented by the *inVRs* framework, illustrated in Figure 6.2.
This model implies that there is a manipulation state (the state in which the user has grabbed an object and can manipulate it) which has to be ended explicitly by the user (for example by pressing a button). To overcome this problem a class was introduced that automatically ends the manipulation state when a space trash object has been collected.

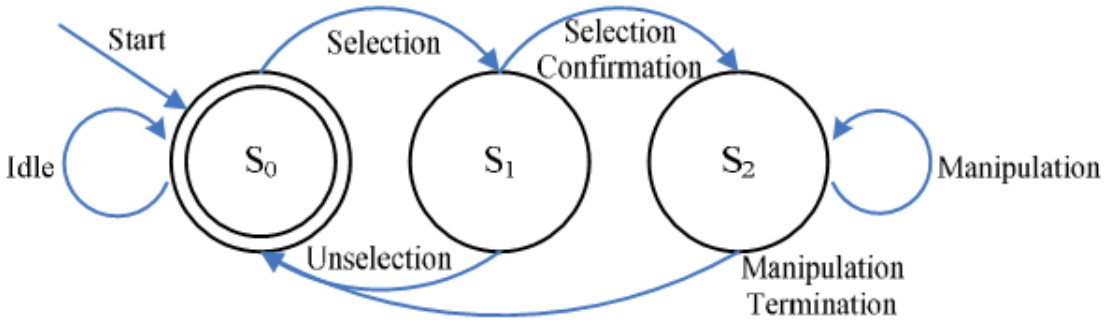

Figure 6.2: Interaction State Machine Implemented in inVRs [AV06]

The modified implementation of the *HOMER* technique consists of the following classes:

- *LimitRayCastSelectionChangeModel*
  This class is used for selecting and unselecting objects. It implements the transition between state S_0 and S_1 in *inVRs*' interaction state machine. An invisible ray extends the

user's grabbing arm. In order to motivate the users to move through the virtual world, the range of the ray is limited and can be easily configured. In case of an intersection of the ray with a trash part that is intended to be collected, the user is notified via the Head Up Display (see Section 6.3) of the selection of the object. Now it is ready to pick.

- *HomerCursorModel*
  A *CursorModel* in *inVRs* is used to calculate the position of the cursor based on the user's real world coordinates. In case of the *HOMER* interaction technique it is additionally used to move the cursor to the trash part after confirming the selection and return it after picking the object. The provided methods for forward and backward movement are called by *PhysicsHomerManipulationModel*.

- *PhysicsHomerManipulationActionModel*
  This is the implementation of the manipulation state S_2. This class is responsible for sending the events used to notify other users and the physics module (see Section 6.4) of the beginning and termination of the manipulation. It pushes the *AutomaticStateTransitionModel* to change the state after manipulation termination.

- *AutomaticStateTransitionModel*
  This class is used to automatically change from manipulation state S_2 into the idle state S_0 after the cursor has returned to its natural position.

6.3 Head-Up Display

Head-Up Displays (or HUDs) are used to display information without distracting the user from the environment. Typical HUDs are found in every situation where the user's attention should remain on the user's main activity, but where some additional information can be useful. Examples are flying airplanes or space ships, driving cars, maintaining equipment that requires directions (measuring tools, weapons etc.) and of course computer games. Typically HUDs display information like speed, direction, position, highlighted targets, levels of limited resources (fuel, oxygen, ammunition,...), crosshairs for aiming etc. Computer game HUDs are a special case of HUDs. On the one hand they simulate HUDs of vehicles, space ships etc. On the other hand they provide the user with information useful for the game (health bars, special capabilities etc). In many cases also ammunition levels, speed, position etc. is displayed even if the simulated entity doesn't display any information about that.
The head-up display of the *Space Trash* game consists of several independent elements that display various information to the user. The basic intention of our head up display is to give the user the feeling of being a pilot in the own space ship. Therefore information like speed, score and target space trash indication needs to be covered. Another main aspect is to guide the pilot to the area that needs to be cleaned from trash. Hence, a mini map is used to show surrounding objects that require the pilots attention.
This section describes which HUD elements exist, what they display and how they are implemented. In detail we will cover the usage of OpenSG for building the HUD elements, and how

they communicate with the corresponding game elements. Additionally important details like the generation of textures containing text (text to texture rendering) will be covered.

6.3.1 OpenSG Usage

InVRs encapsulates 3D objects along with their properties how to address and manipulate them into entities. They can be manipulated using transformations and physical representations. Their current transformations as their representations are maintained in the *World Database* to ease management of them. For further details see [AV06]. However *inVRs* mechanism are very useful for maintaining and manipulating objects inside a 3D scene, but there is no mechanism to statically manipulate on the screen displayed data like for example speed, mini map, score in a HUD would be. HUD elements differ from other entities by the fact that they should always remain on screen, and that their graphical objects are not intended to be manipulated by the physics simulation, user interaction or similar. They are just a visualization of data produced by the game. *InVRs* mechanisms focus more on visualizing and manipulating scenes, and not additional information.
Nevertheless, *inVRs* is designed so flexible that it allows immediate scene graph manipulation, so that programmers can use its facilities where it eases programming without restricting them to use them. To achieve direct scene graph manipulation *inVRs* offers a SceneGraphInterface to access the underlying scene graph. There are possibilities to retrieve the whole scene graph and all the sub scene graphs that represent an entity. The SceneGraphInterface can be accessed using *inVRs* SystemCore. Always-on-screen HUD elements have to be displayed in a static context to the camera's viewport and position. So a good idea is to insert HUD elements in the scene graph below the camera transformation node. This has the advantages that transformations only have to be applied once in relation to the camera transformation. Afterwards these will be updated implicitly by scene graph traversals themselves, the programmer only has to take care of it at one code section.
One thing to cope with is, that the SceneGraphInterface is independent from the underlying scene graph. In our case OpenSG is used. To get OpenSG nodes we have to use this knowledge to tell *inVRs* that we want to use OpenSG explicitly. This is done by using the returned pointer as a concrete OpenSGSceneGraphInterface. Disadvantage of that approach is, that from now on, the application is bound to using OpenSG, it cannot be changed afterwards. This limitation also results from the concrete OpenSG usage afterwards.
All static HUD elements will be added below the camera transformation node. Elements that are bound to entities are added to their sub scene graphs.

6.3.2 Text To Texture

There are basically two ways of generating visual information out of plain text and a mixture between the two.

- text as 3D geometry

- text as texture
- text as mixture between 3D geometry and texture

When using text as 3D geometry in OpenSG the text is rendered into a geometry that consists of polygons. This geometry can be handled like any other node. For example it can be transformed, moved or put into the scene graph.
The second way is creating an image out of textual information. The resulting image is supposed to be an OpenSG image and can be used as texture to prettify geometries.
The last way is a mixture of both techniques mentioned before and renders characters from a font into a texture. This is the approach used in our implementation as it provides flexibility in terms of changing the font and independence from pre-installed found (or not) font files on various operating systems. This is a main advantage because the fancy font used in *Space Trash* is not available on every system it's supposed to run on. The next step towards a texture is to layout the text. Therefore the face has to be layouted by the layout engine which can be parameterized in various ways. After the texture and geometry core are created by using the OpenSG TextFaceFactory it can be put into the scene graph to display the final result. A tutorial how this works in detail can be found in [Abe07]. Figure 6.3 illustrates this method.

Figure 6.3: Texture generated by OpenSG TXF Factory

We used these mechanisms to represent textual data, like speed, score, time, game states etc. to the user.

6.3.3 Target Indicators

The target indicators are used to show the pilots which trash objects need to be collected and to distinguish between trash elements and satellites. The presentation of the indicator differs on the kind of object and whether the object is being targeted by the pilot or not. The following indicators can be found when flying through the world of *Space Trash*:

- No target indicator: Object is not supposed to be collected
- Blue target indicator: Object is a satellite. Pilots are advised to stay away of them!

- White target indicator: Object is in area of interest but out of collecting range
- Red target indicator: Object is a valid trash piece and needs to be collected to gain score.
- Red target indicator (animated): Object is being locked down when targeted. Only locked objects can be collected.

Target indicators are designed as follows:
Basically every object in space (satellite or space trash) holds its own target indicator that is hidden on initial load. The target indicator becomes visible when the object is getting locked by a player or if it is placed within the area that should be cleaned up. Basically the target indicators are implemented by using OpenSG billboards. In order to trigger an animation when the object is being locked, two different billboards are required to work as intended. To show or hide them their sub-scene graphs active flag is set to true or false.

- Target selection indicator
- Target locking indicator

The target locking indicator is implemented as small white circle displaying all space trash objects within the area of interest. If the target is getting locked, the circle will increase in size until it reaches it's intended size.
The target selection indicator is visible as soon as an object enters the user's collecting range. If an object is locked as mentioned above, the target selection indicator will be flipped.
The following code snippet shows a short example of how to use billboards for a head-up display.

```
billboard->setAlignToScreen(true);
billboard->setAxisOfRotation(osg::Vec3f(0,0,0));
billboard->setFocusOnCamera(false);
...
billboardPtr->setCore(billboard);
billboardPtr->addChild(targetIndicatorRotationNode);
billboardPtr->setActive(false);
```

Note that the sticky option within the billboard is the alignment to the screen. Additionally an axis of rotation could be specified to force the billboard to rotate around it. The billboard itself is then used as core for a node pointer (here called billboardPtr) and per default not visible as mentioned above. The transformation node attached to the billboardPtr (targetIndicatorRotationNode) is used for placing the billboard properly to its object.
To implement flipping and resizing objects in OpenSG, transformation nodes and corresponding matrices are used. Additionally an update thread has to trigger the "amount" of size increase or transformation per cycle.An example of how the locking sequence in *Space Trash* looks like is shown below.
The short code example below shows how to modify a nodes' (named tmp) attitude in space. The above mentioned update method has to make sure to deliver adequate values for the rotating angle every time it is called.

```
// flip selection circle
Matrix m;
m.setIdentity();
m.setRotate(osg::Quaternion(Vec3f(1,0,0),angle));
beginEditCP(tmp, osg::Transform::MatrixFieldMask);
tmp->setMatrix(m);
endEditCP(tmp, osg::Transform::MatrixFieldMask);
```

Figure 6.4 illustrated the different states of the target indicators from top left to bottom right showing targets within Area of Interest, satellites, trash pieces in grabber range, and locked trash pieces.

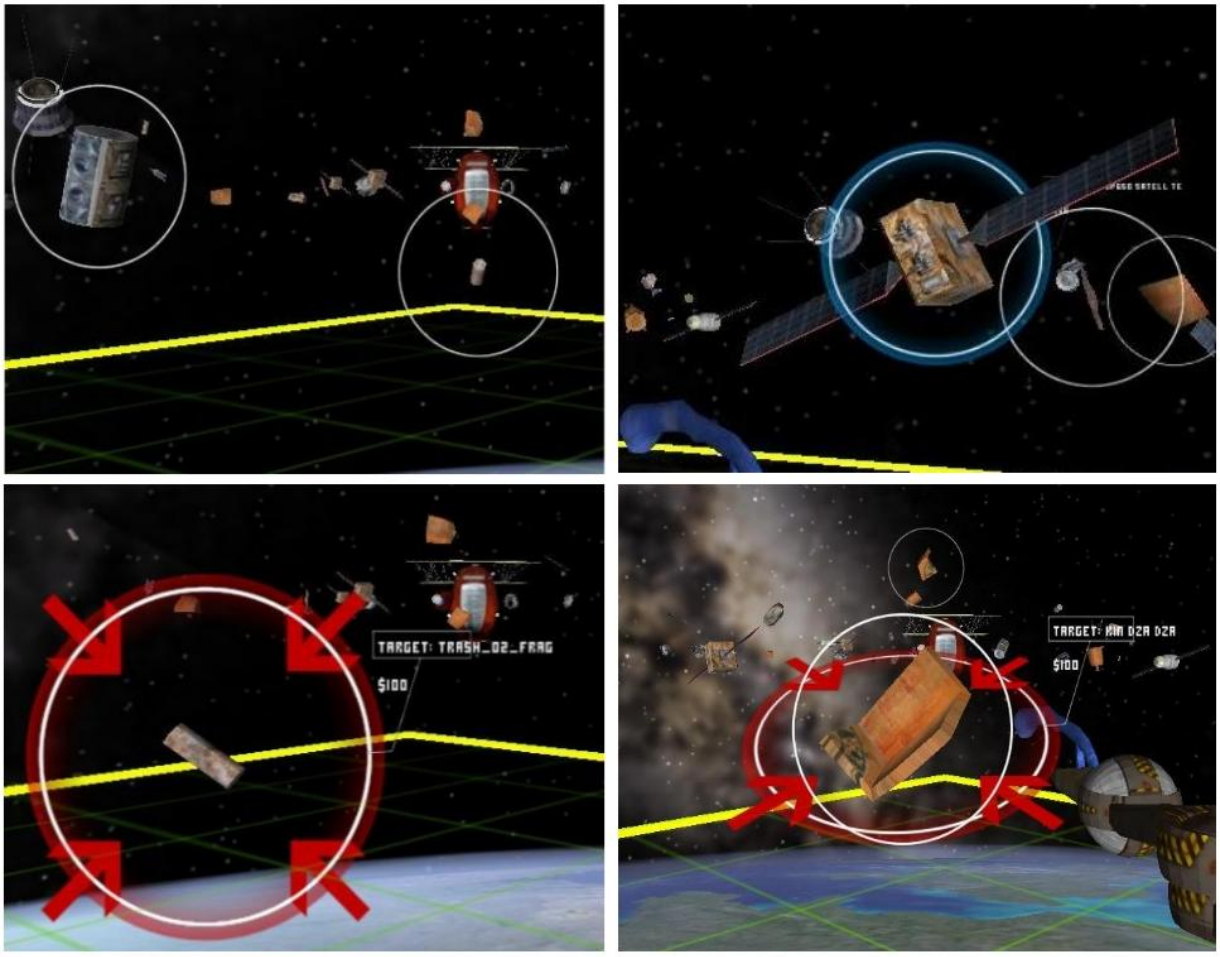

Figure 6.4: Different Markers and States

6.3.4 Area of Interest

The area or environment that needs to be cleaned from trash is highlighted with a thin grid. This environment itself determines which objects either satellites or trash objects, belong to it

and calls the functions required for setting the target indicators to visible or not. The thin grid mentioned before is implemented as a *VRML*-scene and has to be inserted by using the OpenSG SceneFileHandler. The following code snippet shows how to import and use a pre-designed *VRML* model in OpenSG.

```
osg::NodePtr model = NullFC;
model=osg::SceneFileHandler::the().read("../../data/models/Playground.wrl");
...
beginEditCP(myNode);
myNode->addChild(model);
endEditCP(myNode);
```

As one can see that it is fairly easy to add already designed scenes into the scene graph of OpenSG. Figure 6.5 shows such a scene including a playing area which has to be cleaned up.

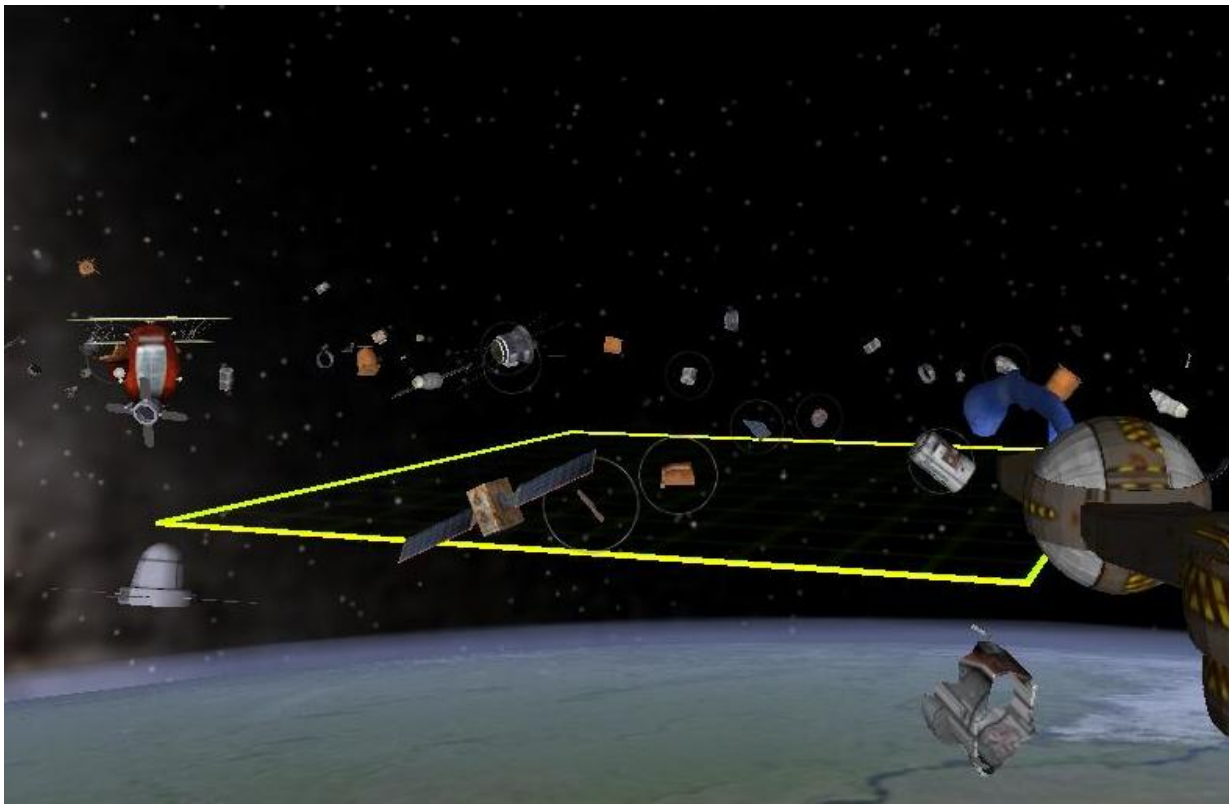

Figure 6.5: Playing Area in Space

6.3.5 Splash Screens

In order to inform the user of the *Space Trash* game in which state the application currently is, a small amount of splash screens are used. They're very important to give the user an idea on what is currently going on. The following game states are covered by splash screens.

- Waiting for other player to join
- Game is starting
- Won game
- Lost game
- Rocket launched

Splash screens are implemented as raw images as the displayed information is static and not generated dynamically by the application. The following code snippet shows how easy it is to implement and display such splash screens in OpenSG.

```
splashScreenStart = osg::Image::create();
beginEditCP(splashScreenStart);
splashScreenStart->read("../../data/maps/Start.png");
endEditCP(splashScreenStart);
addRefCP(splashScreenStart);

beginEditCP(splashScreenMat, osg::SimpleTexturedMaterial::ImageFieldMask);
splashScreenMat->setImage(splashScreenStart);
endEditCP(splashScreenMat, osg::SimpleTexturedMaterial::ImageFieldMask);

beginEditCP(splashPtr, osg::Node::TravMaskFieldMask);
splashPtr->setActive(true);
endEditCP(splashPtr, osg::Node::TravMaskFieldMask);
```

Note that setting masks for node locking is not strictly required in OpenSG but increases performance. After the image has been set to the material node it just has to be activated. After a defined amount of time or a specific event occurred the nodes active attribute is set to false again to hide it. Again the splashPtr node is implemented as billboard so that it always points towards the player. It is fairly important to determine a comfortable distance from the screen to the billboards location when running the application with 3D glasses. This is because the splash screens are rather huge and could cause cybersickness if one appears right in front of the user's eyes. The distance of these nodes in space is determined empirical and might need reconfiguration on different systems.

```
// make sure  splash screen is in front of player
splashScreenOffset = osg::Vec3f(0.0f, 0.0f, -8.0f);
```

The above vector is the offset mentioned before that is very important. As you can see here the splash screen is centered relative to x and y axis but displaced -8 in z direction.
Figure 6.6 shows an example splash screen displayed when a group of players win the game.

Figure 6.6: Example of Splash Screen in *Space Trash*

6.3.6 Mini Map

The mini map is to be found in the middle on top of the 3D scene. It shows a miniature version of the game field to ease orientation and support finding targets. The map is always centered on the player's position, always showing into the view direction of the player. All targets in a certain range are displayed, so that the player sees what's behind the ship. In addition the current area to clear is displayed if in range.
The mini map is a static HUD element which is on screen in every situation. So it is added using the SceneGraphInterface and the camera transformation as described above.
The position of the player's ship is represented as an arrow and two circles which represent the range where targets can be collected.
Trash pieces and satellites are represented as small pyramid shaped objects on the mini map. Trash in range receives a red marker, while satellites are tagged by a blue one. Targets to clear but not in range are just represented as the corresponding pyramid object. This marking behaves similar to the behavior of target indicators. The difference is that there's no selection animation. Figure 6.7 illustrates the possible markers, here one can see two trash pieces in range (red ring)

and one satellite (blue ring).
For a better orientation the mini map also shows the current game field to clear. In Figure 6.7 one can also see how mini map and game field correspond to each other.

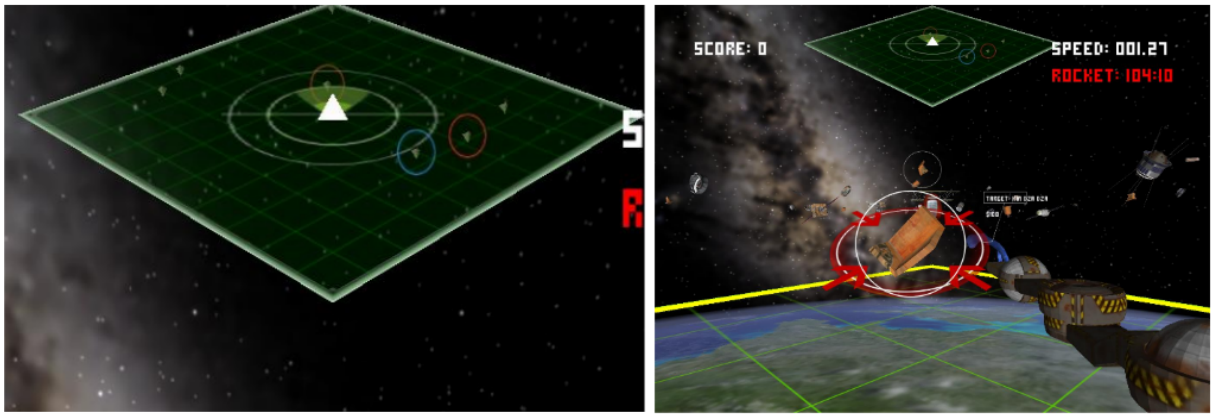

Figure 6.7: Detailed Mini Map and Mini Map over Corresponding Game Scene

The Mini Map is realized as a sub-graph in OpenSG. It is attached as previously discussed in 6.3.1. To keep the visualized data in consistency the HUD main class passes the entities to be visualized as well as the environments which should be represented by the mini map as lists of their World Database Ids to the mini maps [1] update method. This is done every frame. To visualize them in the appropriate scale their coordinates are resized. For that we fixed the size of an environment representation in the mini map to a fixed one, and used this to compute a scale factor between environment and mini map coordinates. We used this scale to apply all the environment translations of the entities to their representations relative to the user's position in the mini map. To get this translation we just used a little bit of vector mathematics: the distance vector between user and the entity to show on the map. Rotation of the map is achieved by using the rotation of the local user's navigated transformation data.

6.3.7 Crosshair

To ease aiming in monoscopic setups we added a green circle to show where the grabber controller is pointing to. This was necessary because only in stereoscopic setups the direction where the grabber controller points to can be guessed using a human's imagination of space. In monoscopic this is not the fact, because the viewport is not considered.
For the implementation we simply used the same raycast-mechanism as for locking targets (see Section 6.2.2). A ray in aiming direction is cast into the scene, starting at the user's position. This ray is then normalized, and resized to a certain length. At the position where the ray ends, the crosshair texture is translated and displayed as a billboard (so that it always points towards the user).

[1] in code "GyroCompass"

6.3.8 Problems Encountered

As mentioned before, we decided to use OpenSG directly to implement the HUD, because *inVRs* doesn't offer facilities to ease creation of HUD elements. *InVRs* abstracts from the underlying scene graph system, which actually holds coordinate systems, OpenGL abstraction, handling of textures, transformations etc. In fact *inVRs* abstraction is a good thing for programmers, because also different scene graph implementations (like OpenSceneGraph) differ in their implementations. But if direct scene graph manipulation is necessary, the programmer has to be aware of the differences.
One example of this differences is found in the used coordinate systems of *inVRs* and OpenSG. One uses z-axis to point into depth, the other one uses z-Axis to point into height. The main problem here was that we mixed coordinates up several times until we were familiar with the different systems.
Doing mathematical calculations we often used the gmtl. The problem here was that gmtl and OpenSG elements, like vectors, coordinates etc. often use the same syntax, method-, object-, fieldnames. In many cases it was not clear if a method returns a gmtl or an OpenSG object. So here we had a lot of trial and error programming.
In some cases we also faced the problem that it was not perfectly clear which coordinates are returned by some methods, like for example the position of a plane or an environment. To determine if it's a corner or a center coordinate triple we also had to use trial and error with adapting coordinates.
After some deeper investigation (debugging etc.), we found out that some mechanisms in OpenSG were not thread safe. So we had to exclude elements to be deleted from scene graph traversal before we were able to delete them.
Besides programming we also had the problem that the application should run on different Virtual Reality installations. Although hardware abstraction is done by *inVRs* and it's underlying and used frameworks, we had the problem that we had to adapt some offsets of HUD-planes and elements, so they won't cause cybersickness. Also we had to find trade-offs in texture sizes, refresh rates etc. so that the HUD would not cause performance issues on any of the machines.

6.4 Physics Simulation

Many current computer games tend to create vivid and lifelike virtual worlds the player can move and interact within. Often the motion behavior of objects in these virtual environments should be similar to the real world. Thus many of these games integrate so called physics engines. A physics engine allows to simulate the behavior of objects according to the laws of physics. Since the exact simulation of object behavior is very complex simplifications have to be made in order to be able to calculate the physics simulation in real-time. The simulation technique which is mainly used in current computer games is called rigid body dynamics.
Rigid body dynamics is used for the real-time simulation of the motion behavior of objects. This kind of simulation deals with rigid bodies, which are objects with an inflexible shape. The simplification of using inflexible shapes reduces the amount of computations needed for the simulation

to an extent that it can be calculated in real-time. The main results of a rigid body dynamics simulation are the positions and orientations of all simulated objects as well as their velocities. The rigid body states are calculated in an iterative way. This means that the calculations are run in a simulation loop and every loop cycle results in the rigid body states at a specific time. In order to achieve smooth object motion this simulation loop has to be executed several times per second.
The rigid body simulation loop cycle can be subdivided into three main tasks: the motion calculation, collision detection and the constraint solving. In the first step the motion of all rigid bodies is calculated. These calculations are mainly based on newton's first and second law. As a result the objects' positions, orientations but also their velocities after each simulation step are obtained. After the new rigid body transformations are calculated the simulation has to check if objects are intersecting. This is done in the collision detection step. In order to recognize interpenetrations of rigid bodies each object must have a pre-defined shape. As a result of this step information which objects are colliding and where the contact points are located are provided. The last task in one simulation loop cycle is to handle the detected collisions and other constraints. Collisions are fixed by applying impulses to the colliding objects. For the solving of constraints force or velocity based methods are used.

6.4.1 Physics Simulation in *inVRs*

The *inVRs* framework provides a physics module which allows for the creation of NVEs with integrated rigid body dynamics. This module is implemented in two parts: the low-level simulation library which calculates the rigid body dynamics and the high-level management layer which manages the synchronization of results and the network communication needed for interaction.
The low-level simulation library is called object oriented physics simulation (oops). It is based on the open source physics engine Open Dynamics Engine (ODE). The library provides an object-oriented interface to the physics engine and additional functionality needed for the execution of the physics simulation.
The central part of oops is the Simulation class. This class manages all simulated objects and provides an implementation of the simulation loop. For the automatic collision response a simple material system is provided by this class which allows for the definition of friction values or other material properties like bounciness.
Besides the physics calculations which are executed by oops the high-level management layer of the *inVRs* physics module has to cope with the distribution of the physics simulation over the network. In the current implementation a single physics server is supported which has to distribute the results of the simulation to all clients. Additionally network messages have to be transmitted if a client wants to interact with a rigid body the server is simulating. This functionality is provided by the Physics class of the *inVRs* physics module. This class uses the simulation loop of oops and extends it by the necessary network communication. Before the execution of the physics calculations the class checks for incoming client input requests and applies these to the rigid bodies. Afterwards the physics simulation step is executed which complies to a single simulation loop cycle. Finally the class calls a registered synchronization model object which then distributes the simulation results to all clients.

For the synchronization of the physics simulation results to the clients different models are provided. Simple models are based on streaming techniques which transfer the resulting transformations of all rigid bodies to the clients. More advanced models use client side prediction in order to reduce the bandwidth consumption for synchronization.

6.4.2 The Physical World of *Space Trash*

In the *Space Trash* application the motion of all entities is simulated by the *inVRs* physics module. This involves all pieces of trash, the satellites as well as the players' space ships. Therefore all entities are extended with a rigid body representation. In order to speed up the physics calculations the used rigid body shapes are simplified approximations of the real 3D models. These approximations are compositions of boxes, spheres and capsules. Figure 6.8 shows an example of a satellite entity and its physical representation which consists of three boxes.

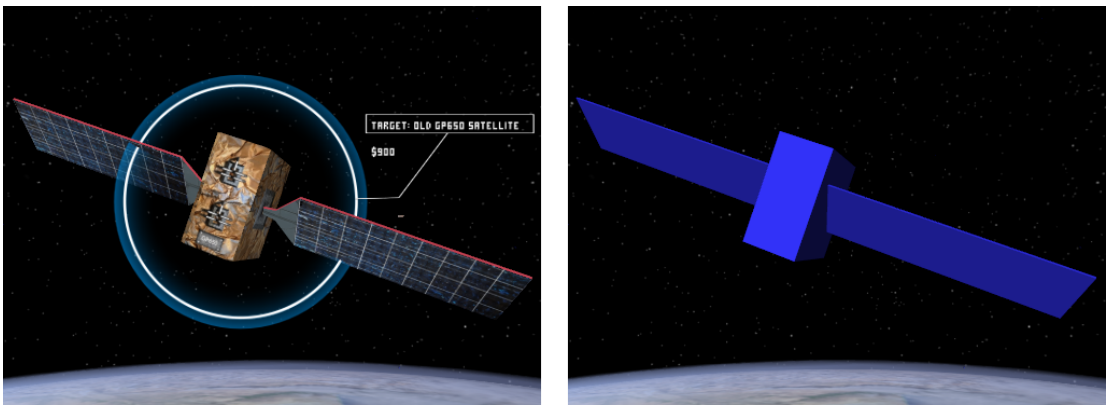

Figure 6.8: Visual and Physical Representation of a Satellite

At the startup of the application each trash piece is given an initial linear and angular velocity in order to achieve a more vivid environment. Because of the lack of friction in a space simulation these objects could leave the playground and get inaccessible to the users. Therefore a manual friction calculation was added to the physics simulation. This friction reduces the velocities of the objects after each simulation step if they are getting above a pre-defined threshold. Furthermore this friction increases the stability of the physics simulation itself since it prevents the objects from reaching high velocities.
A design idea in this application was that trash pieces or satellites can break up into multiple pieces when the user's ship collides with it. To realize this concept in the physics simulation all sub-objects an entity can break into must also provide a physical representation. In the moment a collision between the ship and an destructible object occurs the entity is replaced the sub-entities it splits up to. To avoid that the physical representations of the sub-entities are colliding with each other the collision shapes have to be modelled accordingly. Figure 6.9 shows an example of the physical representation of a full satellite on the left and the representation of the same satellite by its sub-entities on the right side. The physical representations of the sub-entities are modelled slightly smaller to avoid penetrations when the satellite is replaced by its pieces.

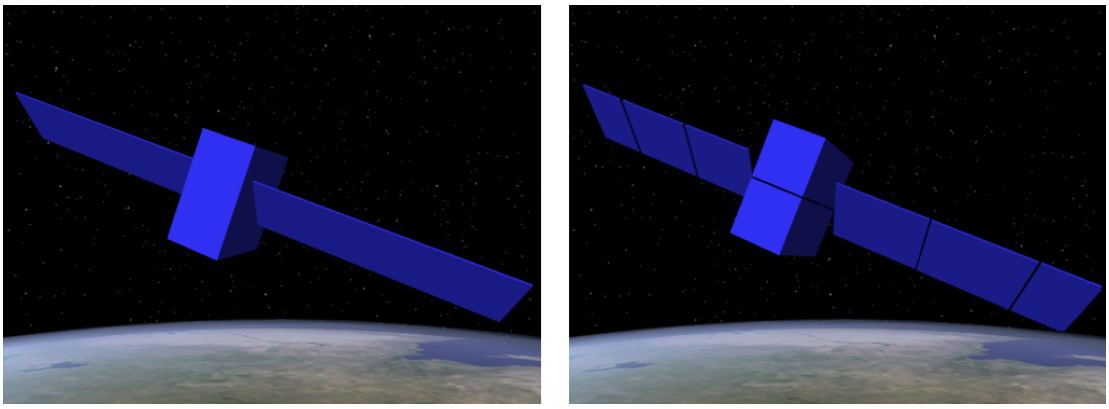

Figure 6.9: Visual Representation of an Intact Satellite (left) and the Destroyed Version (right)

The calculation of the physics simulation is executed on the game server. For the distribution of the results to the clients the FullSynchronizationModel is used. This model streams the positions and orientations of all rigid bodies to all clients at the simulation rate. This produces a high number of network messages but is feasible for a two player scenario. In order to support more players different synchronization models can be used.

6.4.3 Steering the Space Ship

In order to collect the trash pieces in the orbit the users are able to steer their space ships via the input devices. This task is executed by the navigation module of the *inVRs* framework. Usually this module calculates the changes of translation and rotation based on the user's input. This is different in the *Space Trash* application since the user is steering a space ship which is simulated by the physics engine. Calculating the transformation changes for these ships directly in the navigation module would overrule the results from the physics simulation and destroy the smooth movement. Thus the navigation module is configured differently for the *Space Trash* application in order to calculate force and torque values (see Section 6.1). These values are then applied to the rigid bodies representing the space ships which allows the users to accelerate them. One problem in this approach arises because of the lack of friction in the space simulation. Theoretically the users could continue to accelerate the ships until reaching "ludicrous speed". To avoid this the maximum speed of the space ships was limited in the application. This is achieved by checking the linear and angular velocities of the space ships after each simulation step. If the velocities exceed predefined thresholds they are reduced back to these maximum values which prevents the ships from getting to fast.
Another issue arising from the lack of friction is that the navigation of the space ships is getting really complicated when a target should be reached. For example when the users want to fly to a piece of trash they first have to accelerate the ship until they get near the object and afterwards accelerate the ship into the inverse direction in order to reduce the speed of the ship again. Thus the friction calculations which are executed on the trash pieces are also applied on the space ships as well. With these calculations the space ships reduce their speed slightly when no input is applied via the input devices which simplifies the navigation a lot.

6.4.4 Grabbing Trash Pieces

To collect trash pieces the users can invoke their grabbing arm to pick up the pieces as soon as the users are close enough (see Section 7.1). The grabbing arm then moves towards the trash piece until it reaches it. As soon as the target is reached a joint is applied between the trash piece and the grabbing arm. This joint implements a spring and damper system which allows the grabbing arm to pull on the object. As soon as the joint is attached the grabbing arm starts to move backwards to the ship in order to collect the trash piece. To avoid a collision between the ship and the trash piece when it is collected a rule is defined in the *inVRs* physics module which avoids this collision. Thus the grabbing arm can move the trash piece into the ship where it is removed from the application.

6.5 Summary

This chapter has given an insight in the different application components. The mechanisms of interaction and navigation in the context of the *inVRs* framework were explained as well as the used methodologies developed for the *Space Trash* application.
User guidance design has been achieved by creating a threefold HUD, displaying the targets in the scene, the area to be cleared up as well as additional information like the score and speed. To separate the different game phases the HUD makes use of so called splash screens.
A key aspect of the game is the use of physics in the area of rigid body dynamics. Physics is not only important for the lifelike display of space trash but it plays also an important role in the game mechanics. All aspects like interaction, navigation and collisions are dependent on this type of simulation.
In overall all required software components for the *Space Trash* which were not previously available have been described in this chapter.

Chapter 7

Animating *Space Trash*

> In this replacement Earth we're building they've given me Africa to do and of course I'm doing it with all fjords again because I happen to like them, and I'm old fashioned enough to think that they give a lovely baroque feel to a continent. And they tell me it's not equatorial enough. Equatorial!
>
> Slartibartfast - The Hitchhiker's Guide to the Galaxy

Besides sound effects, animations are a common way in computer games to create a more realistic atmosphere, to gain the attention of an user and further to inform an user about actual changes and new situations in the gameplay respectively. Especially, when using virtual reality installations such as a Curved Screen well modelled animations may appear as an additional eye candy to the player or to non-acting observers.
Many elements in *Space Trash* require animations in order to generate a vivid believable virtual world. This chapter introduces the different animation mechanisms which were used to create the application. The simulation of the ships' grabbing arm is based on IK algorithms, while the pre-defined animation sequences make use of the COLLADA file format. Finally, a credits sequence is displayed by a camera following a pre-defined path.
The interaction with picking trash pieces uses the simulation of a grabbing arm which had to be created. Most of the objects in the VE are simulated by the use of a physics engine. All other movements had to be implemented using an animation module, which receives its' animation data from COLLADA files.
In order to display a credits sequence tools for smooth camera motion on given paths were created.

7.1 Grabbing Arm

In *Space Trash* the players navigate a space ship which is equipped with a grabbing arm. The grabbing arm is visualized horizontally centered at the bottom of the screen. One player operates this grabbing arm to collect space trash. Therefore, the player uses the grabber interface (see Section 4.3) to target a single piece of trash in range. The grabbing arm is then moved to the direction of the trash piece so the claw can pick it up and move it out of the scene. This process happens automatically and cannot be interrupted by any user interaction.
The grabbing arm consists of a linked chain of nodes which have a mathematical description (vector) and a visual representation (3D model). Figure 7.1 shows renderings of both grabbing arms – the one from the hacker's ship on the left side and the one from the old men's ship on the right side.

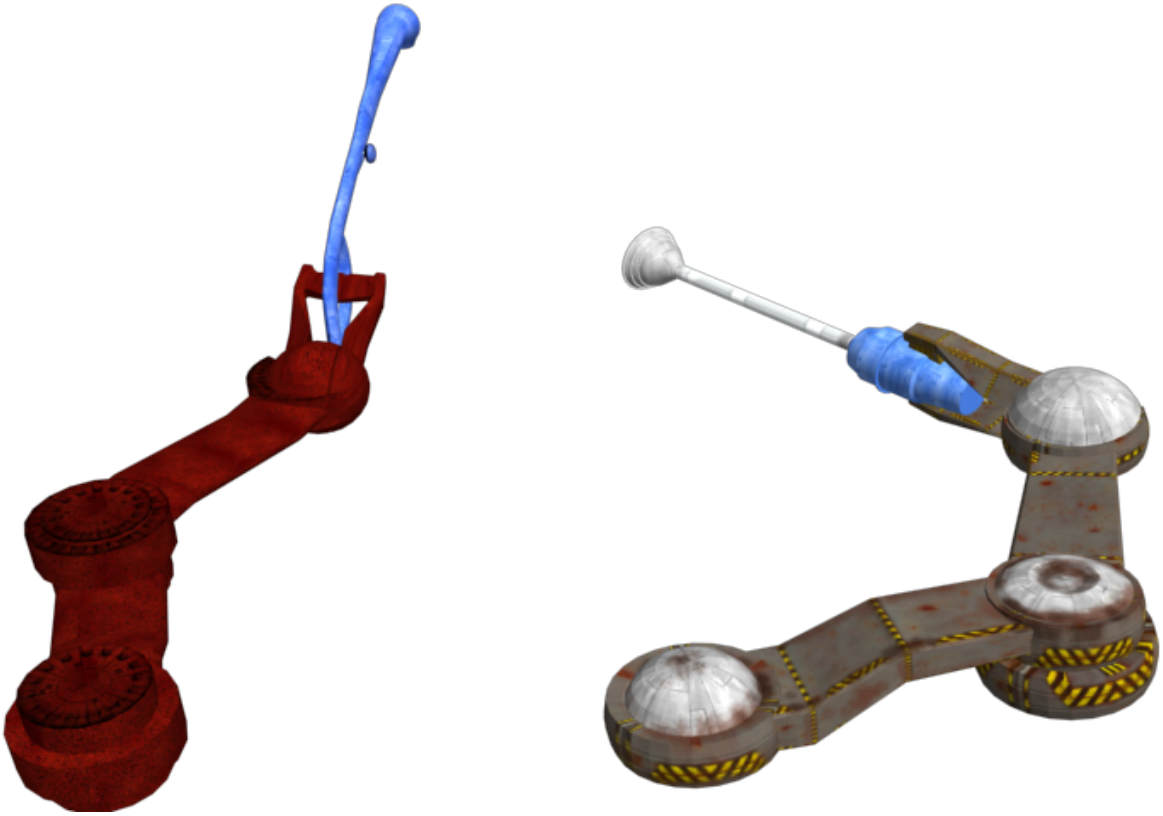

Figure 7.1: Renderings of the grabbing arms

A schematic view of the grabbing arm is shown in Figure 7.2 illustrating the involved nodes.
As it can be seen at the figure the chain consists of two different types of nodes:

- *End-effector* (E):
  An additional node at the tip of the chain for controlling the movement. It completes the chain and defines the length and orientation of the last node in the chain. The *end-effector* is the only node which does not have a visual representation.

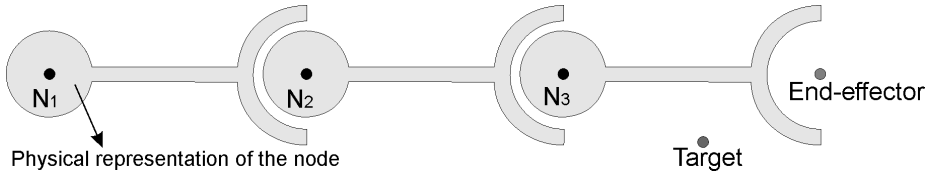

Figure 7.2: Schematic view of the grabbing arm

- Chain node (N_i):
  All other nodes in the chain. The root node (top most parent node) holds absolute position and orientation values within the world coordinate system. All other nodes including the *end-effector* describe their position and orientation relative to their parent.

Movement of the grabbing arm is realized using *inverse kinematics (IK)*. In order to grab a trash piece the *end-effector* of the grabbing arm is moved towards the trash position by rotating the nodes accordingly. The current position and orientation of all nodes is known, as well as the target position. With *inverse kinematics* the rotation values for all nodes are calculated. The opposite of *inverse kinematics* is called *forward* or *direct kinematics*. In *forward kinematics* rotation values of each node are known and thus only need to be applied to the nodes.
From a mathematical point of view *forward kinematics* can be described as a function

$$E = f(N_i)$$

where E is the position of the *end-effector* which can be calculated by the known positions and orientations from the vector of all nodes (N_i) in the chain. In contrast, *inverse kinematics* is an inverse function

$$N_i = f^{-1}(E)$$

where the positions and rotations of all nodes need to be calculated by an inverse function of the *end-effector* position.
Two different approaches were implemented in order to solve the *inverse kinematics* problem: The *Cyclic Coordinate Descent* (*CCD*) method [WC91] and the *Jacobian Transpose (JT)* method [WE84]. Both algorithms are iterative. Hence, at each iteration step the distance between *end-effector* and target is decreased. The break conditions of the calculation loops are the same for either variant, namely:

- Target has been reached:
  This condition is being checked against a certain error threshold which can be set in the configuration file *grabberConfigXX.xml*. The threshold is compared with the length of the vector from *end-effector* to *target* ($\vec{ET}$). More precisely, a squared threshold value is used in order to avoid the need to extract the square root each step.

- Maximum number of repetitions:
  If the target lies outside of the range of the chain, the first condition will never hold. Therefore, the user needs to define a boundary value in the configuration file *grabberConfigXX.xml*.

The *JT* method calculates the current rotation of all nodes in a single step. After applying these rotations the values get updated and influence the next calculation step. In comparison, the *CCD* method calculates the rotation of a single node in one iteration step. Movements with *JT* are more smooth and look more natural in comparison to the *CCD* method. The *CCD* method is easier to understand, simpler to implement and is more efficient in most cases.

7.1.1 Cyclic Coordinate Descent

The *Cyclic Coordinate Descent* method is an iterative approach. At each step the rotation from $\vec{N_iE}$ to $\vec{N_iT}$ is calculated, starting point is node N_n, ending point is the root node N_1. The rotation is applied each step to the current node. The new positions are updated and used for the next step. The implementation is mainly based on [Web02] and [Lan98]. Figure 7.3 shows the rotation of the nodes N_3 and N_2 towards the point T using the *CCD* method. N_1 needs to be rotated also to complete the iteration step.

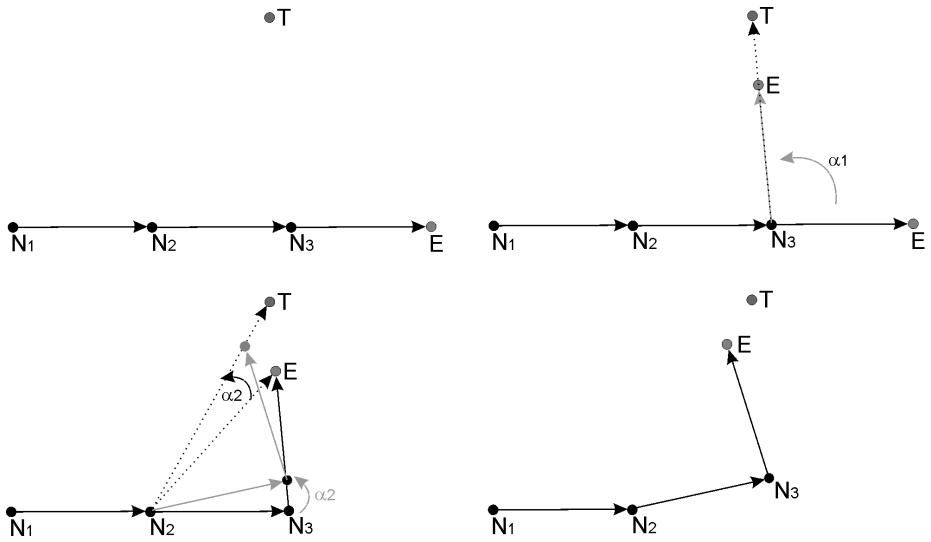

Figure 7.3: Rotation of N_3 and N_2 with the CCD method

It is sufficient to set the number of iterations to 20 with the *CCD* solver in the *Space Trash* application. If the target is reachable, the desired position will be found within 20 loops. Otherwise the chain will be directed towards the target.

7.1.2 Jacobian Transpose

The *Jacobian* matrix describes the relationship between node and the *end-effector* velocities. It is a mapping from input (position and orientation of the *end-effector*) to output (position and orientation of all nodes) values:

$$\dot{E} = J(N_i)\dot{N}_i$$

The following transformation describes the equation which needs to be solved in order to get the joint velocities:

$$\dot{N}_i = J^{-1}(N_i)\dot{E}$$

The *Jacobian matrix* is a *m*x*n* matrix, where *n* is the number of joints and *m* is 6 if one considers the orientation of the *end-effector*, 3 otherwise.
Building the *Jacobi matrix*:

$$J = \begin{pmatrix} \left(\vec{N_iE} \times \vec{rotationAxis_i}\right)^T \\ \left(\vec{rotationAxis_i}\right)^T \end{pmatrix}$$

which is

$$J = \begin{pmatrix} \begin{pmatrix} (\vec{N_1E} \times (\vec{N_1T} \times \vec{N_1E}))_x & \dots & (\vec{N_nE} \times (\vec{N_nT} \times \vec{N_nE}))_x \\ (\vec{N_1E} \times (\vec{N_1T} \times \vec{N_1E}))_y & \dots & (\vec{N_nE} \times (\vec{N_nT} \times \vec{N_nE}))_y \\ (\vec{N_1E} \times (\vec{N_1T} \times \vec{N_1E}))_z & \dots & (\vec{N_nE} \times (\vec{N_nT} \times \vec{N_nE}))_z \end{pmatrix}^T \\ \begin{pmatrix} (\vec{N_1T} \times \vec{N_1E})_x & \dots & (\vec{N_nT} \times \vec{N_nE})_x \\ (\vec{N_1T} \times \vec{N_1E})_y & \dots & (\vec{N_nT} \times \vec{N_nE})_y \\ (\vec{N_1T} \times \vec{N_1E})_z & \dots & (\vec{N_nT} \times \vec{N_nE})_z \end{pmatrix}^T \end{pmatrix}$$

Matrix inversion requires a non-singular and square matrix. These pre-conditions cannot be assumed. Also, the calculation of an inverse matrix causes additional computational load. Therefore, the decision was made to use the *Jacobian Transpose* method which was established by [WE84]. The idea is to replace matrix inversion by matrix transposition[1] which is based on the following relation introduced by Paul [Pau82] as *virtual work*:

$$\tau = J^T F$$

F is a force vector which pulls the *end-effector* towards the target. Normally, the force vector does also consider the twist of the *end-effector*

$$F = \left(f_x, f_y, f_z, \alpha_x, \alpha_y, \alpha_z\right)^T$$

Since only the position of the end-effector is used in the actual implementation, as described in [Spo04], the force vector is

$$F = \left(f_x, f_y, f_z\right)^T \text{ or } F = \left(\vec{ET}\right)^T$$

τ can be seen as representation of the velocities of the nodes, therefore the final formula is

$$\dot{N}_i = J^T F$$

[1]Calculating the *Jacobian* matrix and transposing it, has to be done in each iteration step. In order to save memory access operations these calculations are merged in the implementation, which means, the calculated matrix is stored in its transposed form immediately.

$\dot{N}_i$ holds the rotational values which are further applied to the nodes.
Before applying the rotation values the node velocities need to be integrated:

$$N_i = N_i + h\dot{N}_i$$

This can be done with simple *Euler* integration. As Welman [WE84] states this method is inaccurate for large steps h. Smaller steps will result in more precise results but decrease the performance. He mentions the *Runge-Kutta* method for small steps integration and better performance. The *Jacobian Transpose* implementation uses an *Euler* integration step only.
Spoerl [Spo04] points out that the size of the integration step is very important and is related to the iteration limit. For the *Jacobian Transpose* solver used to control the grabbing arm an iteration limit up to 80 is used, 40 is sufficient for the *Space Trash* application.

7.1.3 Implementation

The *General Math Template Library (GMTL)* is used for all vector and matrix data types respectively and also for trigonometric calculations. In order to avoid gimbal lock issues quaternions are used as the rotation data type. Quaternion data types are also provided by the *GMTL* library. The inverse kinematics library consists of the following classes:

- *IKChainSolver*
  This is the virtual base class for different inverse kinematics implementations.
- *CyclicCoordinateDescent*
  This is the implementation of the Cyclic Coordinate Descent method.
- *JacobianTranspose*
  This is the implementation of the Jacobian Transpose method.
- *ChainNode*
  This class represents a single node of an *IK* chain. It holds the current position and rotation values of a node and provides methods to get these values from a parent-relative or world-relative coordinate system.

The solving system calculates the rotation value for each node and calls *addRotation* in order to get the rotation applied. *addRotation* needs to remove the parents node rotation from the calculated rotation, which is done by the following matrix transformation:

$$relativeRotationM = (absoluteM_{parent})^{-1} * rotationM * absoluteM$$

Constraints of the rotations in all three dimensions can be set for each node in the configuration file. Also the attribute *isConstrained* has to be set in the same file. After solving the inverse kinematics calculation, the responsible class checks whether the applied rotation would be outside the range of the allowed constraint. If the rotational value exceeds the limitation, the rotation will be set to the maximum respectively minimum value. Otherwise, the rotation is applied to the specific node. Both solvers use the same routine for calculating the constraints.

In the *Space Trash* application three independent entities and nodes were created, which means that there is no scene graph hierarchy between these elements. Therefore, the nodes need to get updated with the absolute transformations from *getAbsoluteMatrix*. During the test phase a small *OpenSG* application was implemented where all nodes of the inverse kinematics chain had a hierarchical dependency. In this case *getAbsoluteMatrix* was used to update the root node, but *getMatrix* (returns the transformation matrix relative to its parent) for all other nodes. It is also possible to get rotation values instead of the whole transformation matrix because only rotations need to be applied for the *inverse kinematics*.

7.2 Scripted Animation

In *Space Trash* animations were used to inform the user about new satellites which were added to the playing field. Therefore, a satellite launching animation was created consisting of four basic animation steps: the flight of a satellite-launching rocket towards the playing field, the discharging of the satellite, the opening of the satellite's solar panels and finally the manoeuvring of the satellite to its final destination on the playing field. As there are three different types of satellites the animation of the opening solar panels is specific to each satellite. Figure 7.4 shows the four stages of the animation.
Basically, animations in a networked virtual environment can be implemented in two different ways. On the one hand, animation data (i.e. translation, rotation and scaling values) for each involved scene graph node can be calculated by one peer (e.g. a server) and then sent over network to each client, which in turn applies those data. As an animation may affect a lot of nodes and further many different animations can appear simultaneously this approach might cause a huge amount of network traffic. On the other hand, the information that a certain animation should be started can be sent to each client. Clients then calculate relevant animation data independently from each other. This approach reduces the amount of network traffic to a minimum but may require some additional logic to synchronize animations. In *Space Trash* the second approach was chosen. Implementation details are described in the following.

7.2.1 Implementation

The start of a new animation is initiated by the player (i.e. an entity managed by the game server) which reaches the corresponding state in the gameplay. The server then sends an event to the client which represents the player indicating that a new animation should be started. This client (initiating client) uses the provided parameters from the server – start position of the rocket, detach position and final destination of the satellite – to start the animation locally. This involves the creation of all entities (in terms of *inVRs*) which are part of the animation. It is important to note again that animation-related manipulations essentially affect sub-nodes of an entity and hence are not automatically propagated to clients. Further, the initiating client informs all other clients to start this animation also by sending a corresponding event. Besides the original parameters this event also provides IDs for all animation-related entities. With this information all other clients can run the animation also. When an animation ends the satellite entity is replaced

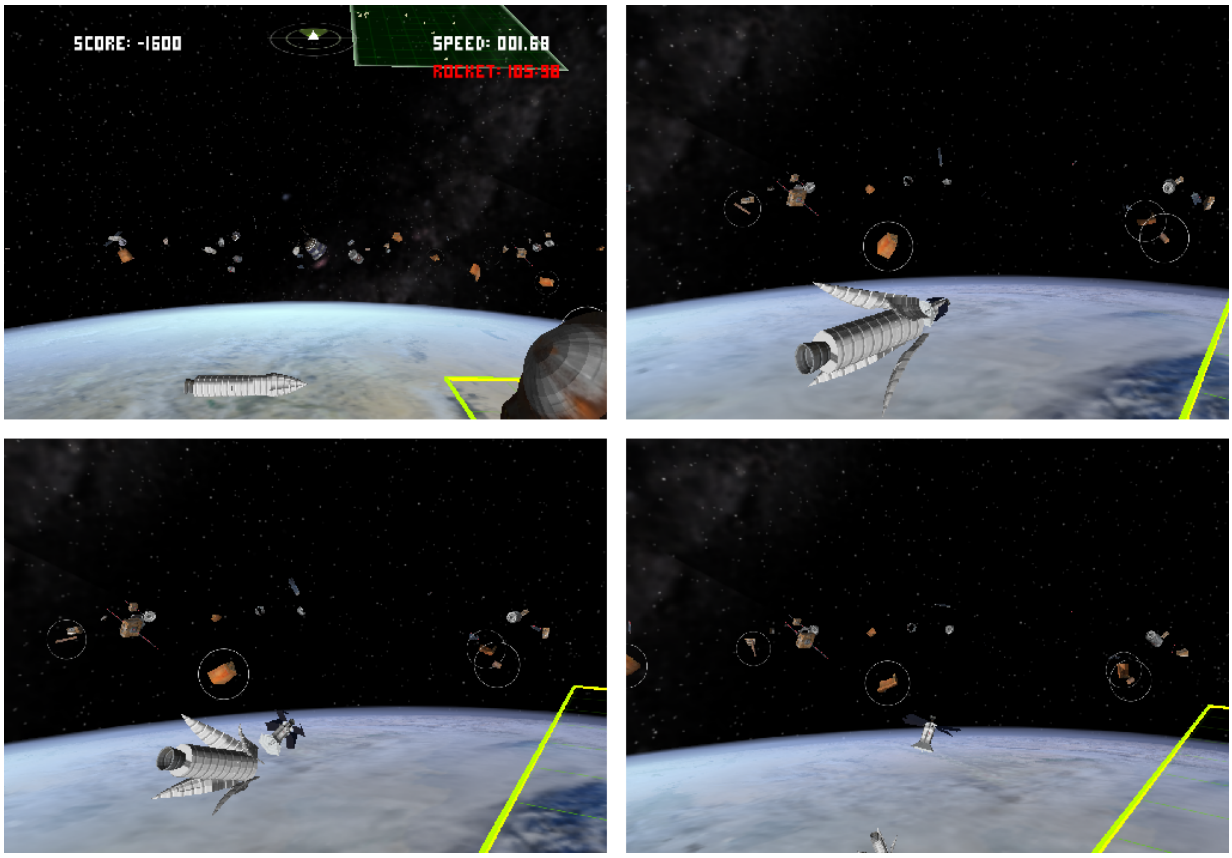

Figure 7.4: Four Stages of the Satellite Launching Animation

by a satellite entity including a physical model. Therefore, the initiating client sends an event to the game server. The game server in turn is responsible for replacing the entity as it manages all game-relevant objects. All other animation-related entities are deleted by the initiating client. A special case arises when a game round ends and animations are still running. If this happens, each client needs to stop the animation. Additionally, the initiating client also needs to destroy all animation-related entities.

For showing animations a separate library was implemented. This library manages models and all containing nodes with its animation-related attributes and uses those data to calculate transformation data for a certain point of time. In *Space Trash* animations were mainly made in *3ds Max*. In order to use those animations in the game they were exported to *COLLADA* DAE files. The animation library thus also supports loading animation-related data from DAE files. Therefore, a *COLLADA* parser was implemented. All other model data are loaded using *inVRs* and *OpenSG* respectively and were stored in VRML files. Nodes are looked up by its name to cross-reference between entity nodes and nodes managed by the animation library.

As stated before, a *COLLADA* parser based on the *COLLADA* DOM API which is provided by www.collada.org was implemented to access data stored in DAE files. This parser implements a class for each single *COLLADA* schema element. This assures easy access to *COLLADA*

schema elements in an object-oriented way. However, only needed core elements and types of the *COLLADA* schema are implemented by now. Further, the parser library takes care of memory allocation and deallocation for handling the rather complex element hierarchies. For more details on the *COLLADA* parser refer to Section 7.3.
The animation library is made up of the following classes:

- *AnimatedModelNode*
  An AnimatedModelNode represents a single node of a complex model and consists of attributes like a unique node name, time data, transformation data, the pivot point and a reference to its parent.

- *AnimatedModelData*
  All AnimatedModelNodes which belong to a certain model are managed by this class.

- *AnimationBuffer*
  The AnimationBuffer is a sigelton class that stores all animations used within a program. It provides methods to load and unload animation data. Currently, loading animation-relevant data from DAE files is implemented. Basically, the sections *library_animations* and *library_visual_scenes* of a DAE file provide relevant information. From section *library_animations* transformation matrices and time values are read; *library_visual_scenes* is parsed to get pivot points for all nodes.
  Besides loading certain data from DAE files, this data also needs to be adjusted. *OpenGL* uses a right-handed coordinate system where the positive x- and y-axes point right and up and the negative z-axis points into the scene, whereas in *3ds Max* the y-axis points into the scene and the z-axis points up (denoted by the schema element *up_axis* in a DAE file). Thus, matrices loaded from DAE files need to be translated to the appropriate coordinate system in this case. Therefore, the transformation matrix needs to be rotated by $90\,^{\circ}$ around the x-axis. Further, y- and z-coordinates of the mesh needs to be exchanged and the new z-coordinate must be multiplied by -1. This is achieved by the following matrix multiplication:

$$transMatrixOGL = xn * transMatrixDAE * xp \tag{7.1}$$

  Where xn is defined as follows

$$xn = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

  and xp is defined as

$$xp = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- *AnimationSolver*
  This class provides methods to start, stop and pause an animation sequence and also calculates transformation data for a certain animation at a certain point of time using previously loaded data. Therefore, at first, the next key frame is determined. The transformation matrix of this key frame is calculated using the following equation:

$$transMatrixFrame = \begin{cases} parentPivot^{-1} * transMatrix * pivot & \text{if parent exists} \\ transMatrix * pivot & \text{otherwise} \end{cases} \tag{7.2}$$

  Then the actual transformation matrix is calculated by interpolating between the last transformation matrix and the current one using the ratio between the current time and the time value of the next key frame as weight factor. Thereby, rotation values are calculated by spherical linear interpolation (slerp). In contrast, interpolation between translations is done with linear interpolation (lerp). Both functions are provided by the *GMTL* library. Interpolation values are only kept for further processing if the transformation is within a given threshold to prevent unnecessary updates of scene graph nodes.

- *EntityAnimation*
  This class represents the interface between an *inVRs* entity and the solver class – it applies given transformation data to *inVRs* entities.

7.3 Parsing COLLADA Files

The purpose of COLLADA in general is to establish an open standard XML database schema that enables 3D authoring tools to freely exchange digital assets. In *Space Trash* it is mainly used as an exchange format for the animation of the satellites as previously described.
The big advantage in comparison to other known formats is, that it allows diverse content processing tools to be combined into a production pipeline. There is no more need for using a specific 3D authoring tool.
Even though the COLLADA DOM Application Programming Interface (API) provides access to the whole feature set of COLLADA, there is still room for optimization. This is especially true if only a subset of the features is needed, for instance just parsing a COLLADA document.
Therefore, an additional layer above the COLLADA DOM was introduced. The purpose of this section is to describe this high-level layer and its usage. Additionally, the techniques which were applied to reduce the learning curve to a minimum while simultaneously maximizing security, are discussed.

7.3.1 Introduction

COLLADA is an acronym for **COLLA**borative **D**esign **A**ctivity. It was originally created by Sony Computer Entertainment as the official format for PlayStation 3 and PlayStation Portable development. Since then it has become the property of the Khronos Group, a not-for-profit technology consortium that now shares the copyright with Sony.
COLLADA defines an open standard XML database schema for an archive-grade format that holds meta information. Due to the nature of XML there is a wide range of compatibility amid different content processing tools. Otherwise, they might store their digital assets in incompatible file formats.
COLLADA documents are usually identified with a .dae (digital asset exchange) filename extension. Many digital content creation (DCC) vendors, commercial game studios and game engines have adopted this standard, such as Google, Alias, Discreet, and Softimage. [2]

7.3.2 The Low-Level Library

The COLLADA DOM (Document Object Model) introduced earlier, is a comprehensive framework for the development of COLLADA applications. It consists among other parts of a C++ library to load, query, and translate COLLADA document content.
The DOM loads COLLADA data into a runtime database consisting of structures that represents those defined in the COLLADA schema. These runtime structures are auto-generated from the schema, and therefore eliminating inconsistency and error. Developers can directly use the data structures loaded into the COLLADA runtime database within their application. Hence, it is quite easy to add import and export capabilities for COLLADA. [3]

7.3.3 The High-Level Library

Often, there is just the need to import COLLADA data structures. For instance a game engine or a viewer application does not need to write new or modified data into the document. For such cases an additional layer above the COLLADA DOM is introduced. This minimizes the time to get used to deal with COLLADA documents.
To make sure working with this high-level interface is as easy as possible, the following guidelines were applied:

- **Simplicity:** The interface is reduced to methods for obtaining data only. Also, dealing with C++ templates is done internally and therefore invisible to the user.
- **Consistency:** A simple naming convention eliminates ambiguity. Every method returns either a const pointer or an elemental data type.

[2] For more information please refer to: http://collada.org/ or http://www.khronos.org/collada/
[3] For more information please refer to: http://collada.org/mediawiki/index.php/DOM_guide:_Introduction

- **Security:** Every memory related action is done internally to eliminate the possibility of memory leaks. Automated testing was applied to make certain the high-level library itself does not cause any memory leaks.

Simplicity

To keep the interface slim and simple, only methods for obtaining data were implemented. This is especially useful if there are code completion tools involved. Once there is a `ColladaItem` object instantiated (see Section 7.3.4), there are solely getter-methods available, enabling the user to navigate top-down in the data structure.
Even though it is necessary to involve C++ templates to use the COLLADA DOM, the high-level interface conceals this for more convenience.

Consistency

For the whole interface the following naming conventions were applied:

- each getter-method has the prefix `get`
- each substring begins with an upper letter
- no underscore separates the substrings
- in cases where the COLLADA schema supports more than one child element, there is a `getXxxCount` method available. The `getXxx` methods then comes with an index as parameter.

For instance, a pointer to the element `<library_visual_scene>` can be obtained via the method `getLibraryVisualScene`. To obtain the number of children from the class `Node`, the method `getNodeCount` is available. Whereas the method `getNode(0)` returns a pointer to the first element `Node`.

Security

To eliminate problems caused by memory leaks and reduce complexity, the library itself takes care of dealing with memory. This means, every single pointer which is obtained via a getter-method is guaranteed to be valid. Of course, the memory will also be freed again automatically, if the `ColladaItem` object is destroyed.
To make certain, all the memory allocations and releases are handled correctly by the library, the classes were extended with functionality to enable the possibility for unit testing. For more information on the subject of automated selftesting please refer to section 7.3.5.
To reduce the possibilities of programming errors due to wrong usage of the library, every pointer returned by a getter-method is declared `const`.
To avoid interference with any other involved modules or classes, the whole library is packed in the `dae` namespace.

Supported Content

Since the high-level library was introduced to support the development of another application, not the whole feature set of COLLADA is supported. The following content is available via the high-level interface:

- **asset** and the most common child elements
- **library_animations** and all child elements
- **library_visual_scenes** and all child elements, except `extra`
- **library_physics_materials** and all child elements
- **library_physics_models** and all child elements
- **library_physics_scenes** and all child elements

7.3.4 A practical Example

Extracting data from a COLLADA document is very straight forward. It just involves a few steps. First, instanciate a `ColladaItem` with the filename of your choice.

```
ColladaItem item(DAE_FILE);
```

From here on you can navigate top-down to get all the data that you might need. Simply request a pointer to any element of interest, and extract attributes or pointers to child elements via the available getter-methods. In the given example the data is just printed out on the console, but obviously, it is quite easy to pass this data to a game engine, for instance.

```
const LibraryPhysicsMaterials *lib = item.getLibraryPhysicsMaterials();

int physics_material_count = lib->getPhysicsMaterialCount();
cout << endl << "lib.PhysicsMaterialCount(): "
    << physics_material_count << endl;

for(int i=0; i<physics_material_count; ++i) {
    const PhysicsMaterial *physics_material = lib->getPhysicsMaterial(i);

    cout << "  physics_material[" << i << "] id: "
        << physics_material->getId() << endl;
    cout << "  physics_material[" << i << "] name: "
        << physics_material->getName() << endl;

    const PhysicsMaterial::TechniqueCommon *technique_common =
        physics_material->getTechniqueCommon();
    const DynamicFriction *dynamic_friction =
        technique_common->getDynamicFriction();
    cout << "  physics_material[" << i << "] dynamic_friction: " <<
```

```
        dynamic_friction->getValue() << endl;
    const Restitution *restitution = technique_common->getRestitution();
    cout << "  physics_material[" << i << "] restitution: " <<
        restitution->getValue() << endl;
    const StaticFriction *static_friction =
        technique_common->getStaticFriction();
    cout << "  physics_material[" << i << "] static_friction: " <<
        static_friction->getValue() << endl;
}
```

7.3.5 Automated Selftesting

Since the parsing process of a COLLADA document involves quite a lot of dealing with memory, there is a good chance, that some allocated memory will not be released again. This so called "memory leaks" can exist without anyone noticing. Nevertheless, they can be responsible for bad performance, especially if the system is short on memory, for instance an embedded device. Without proper debugging skills and a tool designed to detect memory leaks, it can be hard to track them down. Therefore, all classes of the high-level library are equipped with the ability to keep track of their own current amount of instances in the system. This enables the possibility to run unit tests and therefore eliminate potential memory leaks.
After a successfully finished unit test, there is no need to waste performance and memory resources with background activities of the testing process. Therefore, every piece of code related to the unit tests is packed in `#ifdef _TRACEINSTANCES_` preprocessor directives. Thus, at normal runtime there are no side effects of the testing process residing.

Instrumenting Classes

To support the tracking of number of instances, the classes need to be extended with certain functionality. First of all, every class needs to have a private static member variable to count the number of instances. The current value of this counter can be accessed via a static method, and is the same for all classes.
To aid unit tests several macros were implemented:

- `DBG_OUT`: To print debug information in the console. It adds useful information like timestamp, filename and line number. This macro only leads to an output, if the symbols `_PRINTTRACE_` and `_TRACEINSTANCES_` are known at compile time.
- `CTOR`: To increment the instance counter. This has to be added to every single constructor of a class.
- `DTOR`: To decrement the instance counter. Additionally, a check if the destructor isn't called more often than a constructor, will be executed. This has to be added to the destructor.

The header file of such an extended class looks like this:

```
class Foo {
    public:
        Foo();
        ~Foo();

#ifdef _TRACEINSTANCES_
        static int getInstanceCount() { return m_instances; };
#endif

    private:
#ifdef _TRACEINSTANCES_
        static int m_instances;
#endif
};
```

Whereas the source file of such an extended class looks like this:

```
#ifdef _TRACEINSTANCES_
int Foo::m_instances = 0;
#endif

Foo::Foo() {
    CTOR();
}

Foo::~Foo() {
    DTOR();
}
```

Both header and source files of the class `Foo` are available in the package for more convenience.

The Test Class

To make the testing process more serviceable, the helper class `Test` was introduced. It has only three static methods and therefore does not need to be instantiated. The usage of the Test class is illustrated in the following code snippet:

```
Test::Begin(name);
Test::Add(description, condition);
Test::End();
```

So, for any series of tests which should be summarized together, just call the method `Begin`, with a string for the name of the tests. After that, simply add case by case with the method `Add` with a string for the description of the test case, and a boolean for the condition. When all the test cases are added, terminate the test series with the method `End`. A conclusion will be printed afterwards including the number of executed tests and how many of them passed or failed.

The Unit Tests

The unit tests are located in the file `ColladaPOC`. This class implements a `main()`-method, and thus can be compiled into an executable. To run the unit tests, it is necessary to define some preprocessor symbols before compiling:

- `_TRACEINSTANCES_` activates the actual counting of instances. This is mandatory if the unit tests should be done.
- `_PRINTTRACE_` activates the console output and provides additional debug information. This is optional, but it gives detailed information on how and in which sequence the data structure is created.

At this point of time, only one sort of unit test is implemented. Every class is checked, if the appendant destructor is called sufficiently, hence leaving no memory leaks.

7.4 Camera Animation

Sophisticated camera movements along a curve are well known from movies, moving the camera between different planes of depth or following the action during a scene. A similar method finds a common use in cutscenes of computer games, to give a overview of the virtual world. In our application we use a pre-defined camera path to display the credits in between two game sessions. Such a camera path can be defined interpolating between a number of control points.

7.4.1 Hermite spline interpolation

Suppose we have $n+1$ control points $\mathbf{p}_0, \ldots, \mathbf{p}_n$. It is possible to interpolate these points using a parametric polynomial function of the degree n. A disadvantage of this solution is that a polynomial function with a higher degree has enormous inaccuracies and is often in danger to oscillate between control points. We could rather define a polynomial function for every interval separately using boundary conditions to achieve the continuity and join it in a piecewise polynomial curve, called spline. The chosen degree should be a compromise between flexibility, calculation cost and the negative effects of higher degree polynomials mentioned before. To allow for full flexibility the minimal polynomial degree is 3.
We can write a cubic polynomial as

$$\mathbf{p}(u) = \mathbf{a}_3 u^3 + \mathbf{a}_2 u^2 + \mathbf{a}_1 u + \mathbf{a}_0 = \mathbf{u}^T \mathbf{a}$$

where

$$\mathbf{u} = \begin{pmatrix} u^3 \\ u^2 \\ u \\ 1 \end{pmatrix}, \ \mathbf{a} = \begin{pmatrix} \mathbf{a}_3 \\ \mathbf{a}_2 \\ \mathbf{a}_1 \\ \mathbf{a}_0 \end{pmatrix}, \ \mathbf{a}_k = \begin{pmatrix} a_k x \\ a_k y \\ a_k z \end{pmatrix}$$

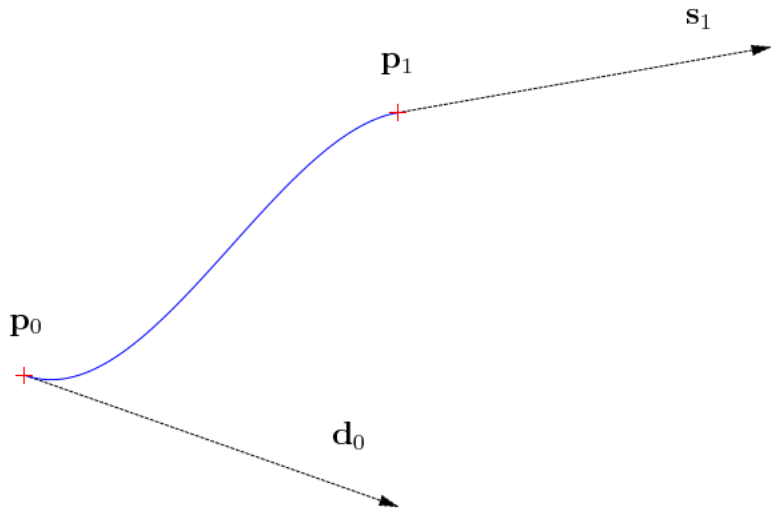

Figure 7.5: Hermite interpolation

The unknown coefficient of the polynomial shall be determined using the control points data and the continuity condition. A class of polynomial interpolations consider the control points and the derivatives at control points to calculate the polynomial called hermite interpolation.
Suppose we want to calculate the hermite polynomial inside the interval $(0, 1)$. This is not a restriction of generality because we can use $u = (t - t_k)/(t_{k+1} - t_k)$ to scale a segment. We get the boundary conditions

$$\begin{aligned} \mathbf{p}_0 &= \mathbf{p}(0) = \mathbf{a}_0 \\ \mathbf{p}_1 &= \mathbf{p}(1) = \mathbf{a}_3 + \mathbf{a}_2 + \mathbf{a}_1 + \mathbf{a}_0 \end{aligned}$$

and

$$\begin{aligned} \mathbf{p}'_0 &= \frac{d\mathbf{p}(0)}{du} = \mathbf{a}_1 \\ \mathbf{p}'_1 &= \frac{d\mathbf{p}(1)}{du} = 3\mathbf{a}_3 + 2\mathbf{a}_2 + \mathbf{a}_1 \end{aligned}$$

which we can write in Matrix form as

$$\mathbf{C} = \begin{pmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}'_0 \\ \mathbf{p}'_1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{pmatrix} \mathbf{a}$$

The solution of this linear system of equations is

$$\mathbf{a} = \mathbf{HC}$$

where $\mathbf{H}$ is the *hermite geometry matrix* [Ang06].

$$\mathbf{H} = \begin{pmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

From this follows the polynomial

$$\mathbf{p}(s) = \mathbf{u}^T\mathbf{HC} = \mathbf{h}(u)\mathbf{C}$$

with the hermite basic functions $\mathbf{h}(u)$ shown in Figure 7.6.

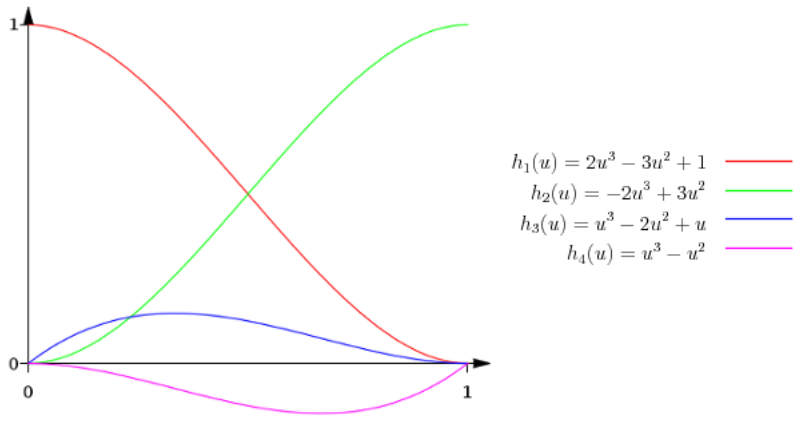

Figure 7.6: Hermite basic functions

Since we want to join the separate polynomial segments to a smooth curve we need some conditions to ensure the continuity. We recognize that

$$\begin{aligned} \mathbf{p}_k(1) &= \mathbf{p}_{k+1}(0) \\ \mathbf{p}'_k(1) &= \alpha \cdot \mathbf{p}_{k+1}(0) \end{aligned}$$

will achieve the requirements. As you can see, the tangents need to point in the same direction, but may have different lengths.
Next we want to know how we can calculate the tangents if we get a set of control points and want to interpolate it with an cubic hermite spline. A simple way is to choose the tangent in the control points to be parallel to the secants between their neighbor points, which is simply the average of the incoming and the outgoing chords.

$$\mathbf{p}'_i = \frac{1}{2}\left((\mathbf{p}_{i+1} - \mathbf{p}_i) + (\mathbf{p}_i - \mathbf{p}_{i-1})\right) = \frac{1}{2}\left(\mathbf{p}_{i+1} - \mathbf{p}_{i-1}\right)$$

This class of cubic splines called *Catmull-Rom splines* [CR74].
If we modify the length of the tangents to affect the tightness of the curve introducing a constant parameter α_i, we obtained the *cardinal splines*.

$$\mathbf{p}'_i = \alpha_i \left(\mathbf{p}_{i+1} - \mathbf{p}_{i-1}\right) = \frac{1-\tau_i}{2} \left(\mathbf{p}_{i+1} - \mathbf{p}_{i-1}\right)$$

The parameter $\tau_i \in [-1, 1]$ in the second view is the tension controls how sharply the curve bends [KB84].
By modifying the trend of the tangents through the introduction of a parameter $\beta_i \in [-1, 1]$ in the equation

$$\mathbf{p}'_i = \frac{1-\beta_i}{2} \left(\mathbf{p}_{i+1} - \mathbf{p}_i\right) + \frac{1+\beta_i}{2} \left(\mathbf{p}_i - \mathbf{p}_{i-1}\right)$$

we can control the bias of the curve.
At that time, both tangents in the control point have the same direction and length. To control the continuity using the parameter $\gamma_i \in [-1, 1]$ we need different equations for the source derivative s_i and the destination derivative d_i of a control point.

$$\begin{aligned}
\mathbf{s}_i &= \frac{1+\gamma_i}{2} \left(\mathbf{p}_{i+1} - \mathbf{p}_i\right) + \frac{1-\gamma_i}{2} \left(\mathbf{p}_i - \mathbf{p}_{i-1}\right) \\
\mathbf{d}_i &= \frac{1-\gamma_i}{2} \left(\mathbf{p}_{i+1} - \mathbf{p}_i\right) + \frac{1+\gamma_i}{2} \left(\mathbf{p}_i - \mathbf{p}_{i-1}\right)
\end{aligned}$$

All three effects can be combined by superposing the equations for tension, continuity and bias. We obtained the *Kochanek-Bartels splines* also known as *TCB splines* announced in [KB84].

$$\begin{aligned}
\mathbf{s}_i &= \frac{(1-\tau_i)(1+\gamma_i)(1-\beta_i)}{2} \left(\mathbf{p}_{i+1} - \mathbf{p}_i\right) + \frac{(1-\tau_i)(1-\gamma_i)(1+\beta_i)}{2} \left(\mathbf{p}_i - \mathbf{p}_{i-1}\right) \\
\mathbf{d}_i &= \frac{(1-\tau_i)(1-\gamma_i)(1-\beta_i)}{2} \left(\mathbf{p}_{i+1} - \mathbf{p}_i\right) + \frac{(1-\tau_i)(1+\gamma_i)(1+\beta_i)}{2} \left(\mathbf{p}_i - \mathbf{p}_{i-1}\right)
\end{aligned}$$

A problem can arise if we use non-uniform sample times for adjacent intervals. In this case a sudden change of speed is produced whenever we pass a control point. To avoid this the tangents must be adjusted to the proportional length [KB84].

$$\begin{aligned}
\overline{\mathbf{s}_i} &= \mathbf{s}_i \cdot \frac{2\Delta t_i}{\Delta t_{i-1} + \Delta t_i} \\
\overline{\mathbf{d}_i} &= \mathbf{d}_i \cdot \frac{2\Delta t_{i-1}}{\Delta t_{i-1} + \Delta t_i}
\end{aligned}$$

A more detailed discussion about that can also be found in [Ebe99].

7.4.2 Camera implementation

There are three points required to describe explicit the position and orientation of an object in the three dimensional space. An obvious choice for a camera is to use the position, view point, and up vector to do this. Each of these points should move along a spline curve. We obtained a very simple prototype containing three splines and the current sample time as data, and functions to load and update the splines.

```
class CameraFly {
    CameraFly(void);
    void load(string file);
    void update(float time);

    HSpline *eye;
    HSpline *view;
    HSpline *up;
    float time;
};
```

If the camera shouldn't roll during the animation, the spline curve of the up vector can be omitted and be substituted by a constant vector.
For our application the camera motion has been created using *Autodesk 3ds Max*, exported in to the 3DS format and is read using the LGPL library *lib3ds*.

7.5 Summary

For the interaction an IK based grabbing arm was developed and described including its mathematical background.
To provide a vivid virtual world objects of the scene have to move. The display of predefined animations as well as the loading of these animations were described in this chapter. An additional camera sequence for the credits part of the game was developed and documented.

Chapter 8

Graphical Effects

> I am Grey. I stand between the candle and the star. We are Grey. We stand between the darkness and the light.
>
> Delen - Babylon 5

To provide the desired visual quality of the initial graphical design several effects had to be developed.

8.1 Shaders

The main utilization for shaders in this installation was the display of shadows of the trash pieces, reflection of metal parts and a realistic representation of the earth. Where for shadows and reflection maps already provided functions of the OpenSG framework could be used, the earth atmosphere had to be implemented by our own shaders. To achieve the maximum compatibility of the shaders to the different deployment platforms we decided to use Cg as shader language and then converted the Cg shaders with the Cg compiler (cgc) to the low-level ARB (OpenGL Architecture Review Board) vertex program and ARB fragment program shaders. These are then added to the objects by adding the FragmentProgramChunk and VertexProgramChunk to a ChunkMaterial assigned to the objects.

8.2 Earthshader

The earth atmosphere graphic effect consists of four layers. The first one is the ground texture layer. With this layer we simulate the earth rotation by simply manipulating the texture coordinates of the surface textures. As we needed high resolution textures of the earth surface and had limited texture size and memory, we decided to split the texture into parts and load them dynamically when they became visible.
The second and third layers are used for the clouds and their shadows. These two layers are

rotating slightly faster than the ground texture in order to achieve the effect of moving clouds. The same dynamic texture loading technique as for the ground textures is used. To save memory a simple grayscale image with alpha values (GL_LUMINANCE_ALPHA) is used for the clouds and their shadows. The shadow layer is colored grey and is slightly displaced to the cloud layer which is colored white. The displacement of these two layers makes the clouds look as they are really floating over the ground. To reduce shader code and enhance performance we pre-calculated the clouds with their shadows in the textures and merged these two layers to one.
Pixel-shader code for the first three layers:

```
float diffuseLight = max(dot(N, L), 0);
float black = pow(1-max(dot(N, E),0),20);
float4 earthcolor;
if(fmod(texCoord.x,2) >= 1)
  earthcolor = tex2D(earth1,frac(texCoord));
else
  earthcolor = tex2D(earth2,frac(texCoord));
float4 cloudcolor;
if(fmod(cloudcoord.x,2) >= 1)
  cloudcolor = tex2D(clouds1,frac(cloudcoord));
else
  cloudcolor = tex2D(clouds2,frac(cloudcoord));
OUT.color = ((cloudcolor*cloudcolor.w + earthcolor*(1-cloudcolor.w))
 *diffuseLight)*(1-black);
```

The fourth layer is for the typical, fog like, blue ozone coloring of the atmosphere as it can be seen in Figure 8.1. Therefore we used the angle between the normal-vector of the earth and the eye-vector to get the border of the seen earth from the observers view. Additionally with the specular-part of the lightning information a slight blue sheen is added to the earth. Where the first three layers are all implemented as a pixel-shader on one simple sphere, the fourth layer is a second, slightly bigger sphere with alpha blending where the atmosphere-shader is implemented as a vertex-shader to enhance the performance.
For the lightning a slightly modified phong shading model [Pho75] is used. To make a smooth crossover from the border of the earth to the space, once more the angle between normal- and eye-vector is used to add a smooth black border to the earth.

8.3 Shadows

By using shadows the spatial impression of the virtual environment is increased. The idea was to set the sun as the main light source which emits light to all the objects (which are mainly trash pieces) that they throw a shadow at the objects behind them. As you can see in the example in Figure 8.2 the visual effect is quite nice. We used the most common technique to create shadows in computer graphics, which is shadow mapping [Wil78]. In shadow mapping, shadows are created by testing whether a pixel is visible from the light source, by comparing it to a z-buffer or depth image of the light source's view, stored in the form of a texture.
Rendering a shadowed scene via shadow mapping involves two major steps. The first creates the

Figure 8.1: Screenshot of the Earth

shadow map and the second applies it to the scene. The first step more or less renders the scene from the light's point of view with the difference to real rendering, that only the depth buffer is calculated. In shadow mapping this depth buffer is also called shadow map. Depending on the type of the light source, an adequate projection type has to be used. For example perspective projection for a point light source and orthographic projection for directional light. If there are multiple light sources, a separate shadow map has to be used for each light source.
In the second step the scene is drawn from the usual camera viewpoint, applying the shadow map. In this process, at first the coordinates of the currently rendering object as seen from the light have to be found. Then those coordinates have to be compared against the shadow map and finally the object has to be drawn either in shadow or in light.
Luckily shadow mapping is already implemented in OpenSG in form of the ShadowMapViewport or the ShadowViewport but we had to do some changes in the CAVESceneManager to make it work. Normally the CAVESceneManager just creates a viewport for mono applications and two instances of the StereoBufferViewport for stereo applications. In order to use the shadow functionality of OpenSG, instances of the ShadowViewport have to be used. The ShadowViewport controls the shadowmap settings and performs the rendering needed for the shadows. Additionally to changing the viewport creation process, we also had to add a whole set of methods to enable the users of the CAVESceneManager to configure the ShadowViewport.
A ShadowViewport has the same settings as a viewport or StereoBufferViewport plus additional

shadow settings. Considering the size of the shadowmap, it was difficult to choose a value which leads to good shadow quality but doesn't drain too much performance. The ShadowViewport supports six different shadow modes as illustrated in Figure 8.2 [1]:

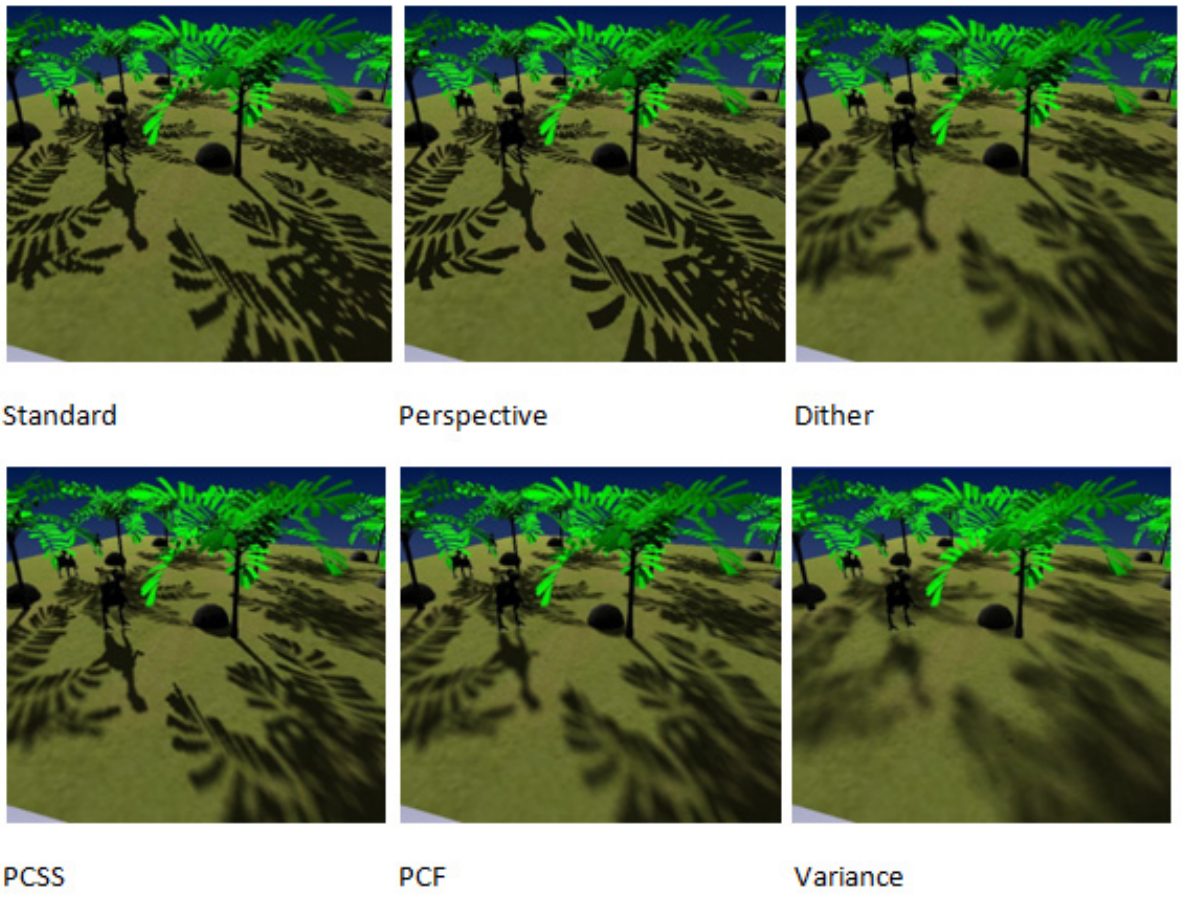

Figure 8.2: Shadow Modes

We used the standard mode as you can see in the following code demonstrating the ShadowViewport setup.

```
ShadowViewportPtr svp = ShadowViewport::create();
beginEditCP(svp);
svp->setSize(size[0],size[1],size[2],size[3]);
svp->setCamera(camDec);
svp->setBackground(bg);
svp->setRoot(_internalRoot);
svp->setLeftBuffer(left); // for stereo
svp->setRightBuffer(!left); // for stereo
svp->getForegrounds().push_back(_foreground);
```

[1] Taken from http://opensg.vrsource.org/trac/wiki/Gallery/Shadows

```
//shadow
svp->setOffFactor(4.0);
svp->setOffBias(8.0);
svp->setGlobalShadowIntensity(0.9);
svp->setMapSize(1024);
svp->setShadowMode(ShadowViewport::STD_SHADOW_MAP);
//svp->setShadowColor(Color4f(0.1,0.1,0.1,1.0));
svp->setShadowSmoothness(0.0);
endEditCP  (svp);
```

After initial setup the ShadowViewport needs to know where the light sources are. They can be added or removed as shown by the following statements.

```
svp->getLightNodes().push_back(light); // add a light node
svp->getLightNodes().erase(svp->getLightNodes().find(light));
```

Some objects should not throw a shadow e.g. HUD elements. You can set this through the following statement.

```
svp->getExcludeNodes().push_back(exclude);
```

8.4 Reflection Maps

Because in reality metal pieces tend to reflect light like mirrors and nearly all the objects in *Space Trash* are made of metal, some technique to simulate this effect is needed. In computer graphics, reflection mapping [BN76] is an appropriate method. It efficiently simulates complex mirroring surfaces by precomputed texture images. The texture is used to store the image of the environment surrounding the rendered object. Using multi texturing the texture is drawn at the surface of the object. There are multiple types of reflection mapping depending on the form of the texture (cubic, spherical). And the texture can either be calculated in real-time or prior run time, which of course disables the reflection to react to changes in the environment but saves performance.
Reflection mapping is directly supported by OpenSG. We used a cubic reflection map but OpenSG supports spherical reflection maps too. So we setup reflection map via using a CubeTextureChunk, which is responsible for the texture and a TexGenChunk, which is responsible for the texture coordinate generation in order to map the texture at the object. Our self-implemented method setReflRec traverses the node tree of the given node modelNode and adds the reflTex and texgen to all the cores which are either of type Geometry or MaterialGroup. So if you use this method on the root node of a model, the reflection map given by reflTex and texgen is attached to it.

```
CubeTextureChunkPtr reflTex = CubeTextureChunk::create();
beginEditCP(reflTex);
reflTex->setImage(reflImage0);
reflTex->setPosXImage(reflImage1);
reflTex->setPosZImage(reflImage2);
reflTex->setNegXImage(reflImage3);
```

Figure 8.3: *Space Trash* without (left) and with (right) reflection mapping. You can see the reflection of the moon on the canvas of the satellite on the right image.

```
reflTex->setPosYImage(reflImage4);
reflTex->setNegYImage(reflImage5);

reflTex->setEnvColor(Color4f(0, 0, 0, 0.2));
reflTex->setEnvMode(GL_COMBINE);
reflTex->setEnvCombineRGB(GL_INTERPOLATE);
reflTex->setEnvSource0RGB(GL_TEXTURE);
reflTex->setEnvSource1RGB(GL_PREVIOUS);
reflTex->setEnvSource2Alpha(GL_CONSTANT);
endEditCP(reflTex);

TexGenChunkPtr texgen = TexGenChunk::create();
beginEditCP(texgen);
texgen->setGenFuncS(GL_REFLECTION_MAP_ARB);
texgen->setGenFuncT(GL_REFLECTION_MAP_ARB);
texgen->setGenFuncR(GL_REFLECTION_MAP_ARB);
endEditCP(texgen);

setReflRec(modelNode, reflTex, texgen);
```

8.5 Summary

All in all the main problem we had with the graphical effects was the performance. Using shadows drained the performance on the PRISM by half so we couldn't activate it in the first installation.
And as the reflection maps were only a "nice to have" feature the time was too short to integrate it in the end.
The atmosphere shader on the other hand was important for the realistic look of our simulation, so this was the only effect that we successfully integrated in the installation at the European

Researchers' Night. But we also had to make some compromises to finally get a fluently running application. We hope that we can integrate more effects in later versions of *Space Trash*.

Chapter 9

Audio

> Life has a melody, Gaius. A rhythm of notes which become your existence once played in harmony with God's plan.
>
> Number Six - Battlestar Galactica

Audio is often used in the area of VEs to increase the degree of immersion. An additional reason for audio in such environments lies in the sonification. For important events or interaction is is very common to support the user by providing additional audio feedback. *Space Trash* uses audio to create the atmosphere of the game by playing a soundtrack. Sound effects are incorporated to create inter action and game play feedback.

9.1 Soundtrack

The soundtrack consists of four separate tracks – *credits loop, waiting loop, main soundtrack, high scores loop* – each of which can be switched more or less seamlessly to its successor. Especially the time of transition between the waiting loop and the main soundtrack cannot be predefined and must therefore be held flexibly. This is achieved by avoiding rhythmic structures in the waiting loop, using the same harmonic basis and starting the main soundtrack with a bang consisting of a piano and bass drum hit to cover potential acoustic artifacts caused by cropping of the sample playback. Throughout the whole soundtrack there are low frequency vibrations to create an unfamiliar atmosphere that the player cannot only hear but also feel.
The credits loop contains only one static pad and some flanged machine-like sounds that puts the player into a journey through space. Additionally, there are "swooshes" synchronized to the camera movements in the video. The waiting loop is even more reduced to only the static pad and the low frequency vibrations. As mentioned above, when the players decide to start the game, the main soundtrack then begins with a bang and a large reverb. There is a lot of reverb and echo on all melody instruments to suggest the feeling of infinite space and to keep everything wide and distant. Furthermore, the use of reversed guitar and piano samples represents the idea of the relativity of time. As the game can be split in three sections – a warm-up phase, the start of the

first satellite and the start of the second satellite – the structure of the soundtrack is analogue to this. The warm-up phase is more of an atmosphere with harmonic elements than a real song, with very less rhythmical order and slow movements to let the player float freely through space. When the first satellite is started, the song begins to develop and the first concrete rhythmical component comes in – in an unusual 10/4 time to pursue the extraordinary mood. Next a gated and dirty drum beat joins to illustrate the trash particles. From the time on where the second satellite is started, the song builds up to a climax, pushing the player's tension until it all breaks down with a bang again. The sound then dies away into the high scores loop, which is the same as the credits loop only without the swooshes so that it can easily be blended into the credits loop again.

9.2 Sound Effects

For the fact that one wouldn't hear any sound in outer space there is no physical reason to provide sound effects. But from the gameplay's point of view it is still useful to have still a few sound effects that give feedback to the player. This is the case for the movement of the grappler and collisions. For the grappler, an ordinary hand blender was recorded and pitched. The collisions were done with plastic and glass bottles and some post-processing. Since the effects sounded very direct and close when they were played as dry samples, again a large reverb was added to create the impression of space and distance.

9.3 Summary

The two aspects of the audio for the *Space Trash* application were briefly introduced in this chapter. An overview on the composition of the soundtrack matching the different game phases was given. For additional feedback to the user the sound effects provide support. Their functionality and creation was described.

Chapter 10

Conclusions

> I've seen things you people wouldn't believe. Attack ships on fire off the shoulder of Orion. I watched C-beams glitter in the dark near the Tannhäuser Gate.
>
> Roy Batty - Blade Runner

A huge variety of installations from the domain of new media art exist but most of them are not at all or just barely documented. This report provides an in depth description of one of such installations – *Space Trash*.
An interdisciplinary group of researchers and artists created an interactive installation, combining the knowledge and experience from their domains. Interface design, visual design, game design, 3D modeling were combined with Human Computer Interaction (HCI) experience, VR display hardware, as well as many aspects from software development like rigid body dynamics simulation, networked communication and general software architecture in order to create the *Space Trash* application and installation.
The previous chapters have introduced and described in detail the inner workings of the *Space Trash* application. Aspects like game design and graphical design were presented and the internal architecture with its software components was described. To develop a truly immersive and vivid virtual world graphical effects had to be created and a soundtrack accompanying the game play with additional sound effects was composed. One of the key aspects of *Space Trash* are the novel input devices which are used to collaboratively control a space ship at each of the interconnected sites.
This chapter will describe the setup and the presentation approach of the prototype implementation which was presented in September 2008. Conclusions from the design and development process will be drawn and finally an outlook into future developments and enhancements is given.

10.1 Installation

The prototype of the *Space Trash* application was demonstrated on the 26th of September 2008 at the European Researchers' Night 2008 at the Johannes Kepler University and the University of Art and Industrial Design. Both sides followed a significantly different presentation approach looking at the technical or on the artful aspects.

10.1.1 Setup at the Johannes Kepler University

To demonstrate the application to a large audience panoramic displays like the i-Cone™ [SG02] are ideal. The installation setup made use of the Virtual Reality Center's (VRCs) Curved Screen, which is such a panoramic display consisting of three active stereo projectors including hardware edge blending.

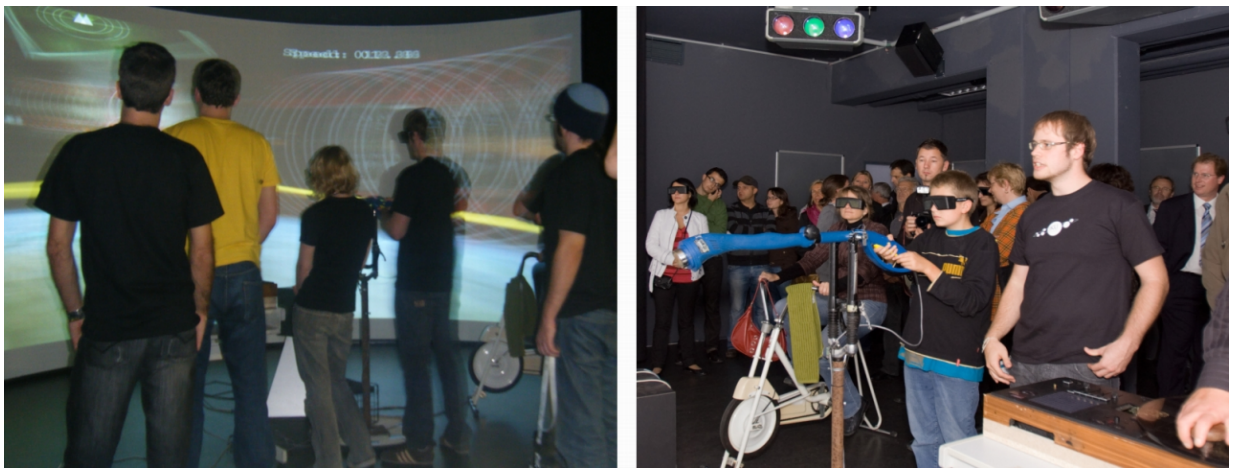

Figure 10.1: The *Space Trash* Application at the Johannes Kepler University

Figure 10.1 shows the audience interacting with *Space Trash* in front of a curved display. The left side of the illustration shows a team of developers evaluating their game, while the right side shows the application being demonstrated at the European Researchers' Night.

10.1.2 Setup at the University of Art and Industrial Design

The setup presented at the University of Art and Industrial Design consisted of a powerwall display using two Digital Light Processing (DLP) projectors for the passive stereoscopic display on a back-projected screen. The advantage of having back projection allowed to set up the devices closer to the screen, while with front projection the users might cast shadows in the scene.
The presentation at the University of Art and Industrial Design, Linz was focused on the design issue and the artistic concept of the project rather than the used technology. We have posted fictional recruit poster for space trash collectors to transport the background story of the game

without making verbal explanation. Additionally, there was a fictional advertisement movie presented originating from the recruiting corporations that are played on a separate screen in the venue. The players had to wear cowboy hats, which is not directly a part of the game play, but played a role in the game experience.

Figure 10.2: Posters at the University of Art and Industrial Design

Figure 10.2 shows some example posters displayed at the University of Art and Industrial Design.

10.1.3 Setup at Laval Virtual

The application was presented at Laval Virtual in Laval, France in April 2009. Two directly adjacent screens were used for the display of *Space Trash*. The screens were back-projected using monoscopic output. An interesting aspect about this setup was the possibility to directly perceive the behavior of the opponent team.

Figure 10.3: The *Space Trash* Application at Laval Virtual

Figure 10.3 illustrates the setup used at Laval Virtual.

10.1.4 Alternative Setups

Depending on the available display hardware a variety of setup possibilities in larger or in smaller scale arise. It is thinkable to setup the application in a truly immersive device like the CAVE™ [CNSD+92]. But ideally multi-display projections walls should be used.
The minimum setup works is a single site fashion, allowing three different users to control the spaceship and to collect trash pieces while running against the time. Stereoscopic display is not considered a necessary requirement, but it allows to significantly increase the feeling of immersion and the illusion of flying through space.
In general it is possible to work with a single projector using either front or back projection. The projection distance and thus the resulting size of the installation is highly dependent on the available projectors. Having a back projection setup would lead to the advantage that the user could move closer to the screen, which carries in our specific case a rather low relevance. This is due to the needed aiming distance of the grabber which cannot be reduced. When using larger screen areas the distance from the grabber to the front screen has to be sufficiently large in order to aim and target at every possible corner of the display.
Using conventional Desktop TFT screens for display is not reasonable due to the grabbing interface which is used to point at the projection wall.

10.2 Future Work

The technology developed for this installation could be used as well in other areas, where for example multiple participants control vehicles or other entities in a networked environment.
The development of a desktop version would improve the possibility to present the storyline to the audience with an introductory sequence, since the type of presentation can be altered. Although this will change the balance between rules, the fictional world and user interaction. It will be interesting to study the difference considering the game experience. Game play refinement would introduce different difficulty level and balance out the different advantages and disadvantages considering the different types of interfaces and displays.
Future user studies and evaluation will provide interesting insights on the communication inside the teams participating and collaborating. It would be interesting to evaluate task performance restricting the participants communication channels (e.g. no talking allowed).

Bibliography

[Abe07] Oliver Abert. Opensg starter guide 1.8.0, 2007. 42

[AHKV04] Christoph Anthes, Paul Heinzlreiter, Gerhard Kurka, and Jens Volkert. Navigation models for a flexible, multi-mode vr navigation framework. In *ACM SIGGRAPH on Virtual Reality Continuum and Its Applications in Industry (VRCAI '04)*, pages 476–479, Singapore, June 2004. ACM Press. 36

[ALBV07] Christoph Anthes, Roland Landertshamer, Helmut Bressler, and Jens Volkert. Managing transformations and events in networked virtual environments. In *ACM International MultiMedia Modeling Conference (MMM '07)*, volume 4352 of *Lecture Notes in Computer Science (LNCS)*, pages 722–729, Singapore, January 2007. Springer. 23, 37

[Ang06] Edward Angel. *Interactive Computer Graphics*, volume Fourth Edition. Addison Wesley, international edition edition, 2006. 71

[ard] Arduino. www.arduino.cc. 17

[ASW+09] Christoph Anthes, Mika Satomi, Alexander Wilhelm, Christa Sommerer, and Jens Volkert. Space trash an interactive networked virtual reality installation. In *Virtual Reality International Conference (VRIC '09)*, Laval, France, April 2009. 3

[AV06] Christoph Anthes and Jens Volkert. invrs - a framework for building interactive networked virtual reality systems. In Michael Gerndt and Dieter Kranzlmüller, editors, *International Conference on High Performance Computing and Communications (HPCC '06)*, volume 4208 of *Lecture Notes in Computer Science (LNCS)*, pages 894–904, Munich, Germany, September 2006. Springer. 2, 21, 36, 39, 41, 90

[AWL+07] Christoph Anthes, Alexander Wilhelm, Roland Landertshamer, Helmut Bressler, and Jens Volkert. Net'O'Drom – An Example for the Development of Networked Immersive VR Applications. In Yong Shi, Geert Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *International Conference on Computational Science (ICCS '07)*, volume 4488 of *Lecture Notes in Computer Science (LNCS)*, pages 752–759, Beijing, China, May 2007. Springer. 1

[BH97] Douglas A. Bowman and Larry F. Hodges. An evaluation of techniques for grabbing and manipulating remote objects in immersive virtual environments. In *ACM Symposium on Interactive 3D Graphics (SI3D '97)*, pages 35–38, Providence, RI, USA, April 1997. ACM Press. 37, 38

[BN76] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, October 1976. 78

[CNSD+92] Carolina Cruz-Neira, Daniel J. Sandin, Thomas A. Defanti, Robert V. Kenyon, and John C. Hart. The cave: Audio visual experience automatic virtual environment. *Communications of the ACM*, 35(6):64–72, June 1992. 24, 86

[CR74] Edwin E. Catmull and Raphael J. Rom. A class of local interpolating splines. In Richard F. Riesenfeld Robert E. Barnhill, editor, *Computer Aided Geometric Design*, pages 317–326. Academic Press, Inc. Orlando, FL, USA, March 1974. Proceedings of a Conference Held at the University of Utah, Salt Lake City, Utah. 72

[Ebe99] David Eberly. Kochanek-bartels cubic splines (tcb splines). Technical report, Geometric Tools, LLC, 1999. 72

[Fra07] Gonzalo Frasca. *Play the message: Play, game and videogame rhetoric*. PhD thesis, IT Univ. Copenhagen, Denmark, 2007. 16

[Juu05] Jesper Juul. *Half-Real: Video Games between Real Rules and Fictional Worlds*. MIT Press, 2005. 7

[KB84] Doris H. U. Kochanek and Richard H. Bartels. Interpolating splines with local tension, continuity, and bias control. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 33–41, New York, NY, USA, 1984. ACM. 72

[Kir08] Friedrich Kirschner. Lecture notes from the games workshop at interface culture, institute of media studies, university of art and industrial design linz,. http://www.zeitbrand.de/, 2008. 5

[Lan98] Jeff Lander. Making kine more flexible. *Game Developer Magazine*, November 1998:15–22, November 1998. 57

[Pau82] Richard P. Paul. *Robot Manipulators: Mathematics, Programming, and Control*. MIT Press, Cambridge, MA, USA, 1982. 58

[PBWI96] Ivan Poupyrev, Mark Billinghurst, Suzanne Weghorst, and Tadao Ichikawa. The go-go interaction technique: Non-linear mapping for direct manipulation in vr. In *ACM Symposium on User Interface Software and Technology (UIST '96)*, pages 79–80, Seattle, WA, USA, November 1996. ACM Press. 37, 38, 90

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975. 75

[ps2] Ps2 mouse interface. http://www.arduino.cc/playground/ComponentLib/Ps2mouse. 18

[Rei02] Dirk Reiners. *OpenSG: A Scene Graph System for Flexible and Efficient Realtime Rendering for Virtual and Augmented Reality Applications*. PhD thesis, Technische Universität Darmstadt, Mai 2002. 2

[SG02] Andreas Simon and Martin Göbel. The i-cone - a panoramic display system for virtual environments. In *Pacific Conference on Computer Graphics and Applications (PG '02)*, pages 3–7, Beijing, China, October 2002. IEEE Computer Society. 24, 84

[Spo04] Marco Spoerl. *Game Programming Gems IV*, chapter The Jacobian Transpose Method for Inverse Kinematics, pages 193–204. Charles River Media, 2004. 58, 59

[Sut68] Ivan E. Sutherland. A head-mounted three-dimensional display. In *Fall Joint Computer Conference AFIPS Conference*, pages 757–764, Fall 1968. 24

[SZ03] Katie Salen and Eric Zimmerman. *Rules of Play – Game Design Fundamentals*. MIT Press, 2003. 6

[WC91] Li-Chun Tommy Wang and Chih Cheng Chen. A combined optimization method for solving the inverse kinematics problem of mechanical manipulators. *IEEE Transactions on Robotics and Automation*, 7(4):489–499, 1991. 56

[WE84] W. A. Wolovich and H. Elliot. A computational technique for inverse kinematics. In *Proceedings of the 23rd IEEE Conference on Decision and Control*, volume 23, pages 1359–1362, December 1984. 56, 58, 59

[Web02] Jason Weber. *Game Programming Gems III*, chapter Constrained Inverse Kinematics, pages 192–199. Charles River Media, 2002. 57

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *SIGGRAPH 78*, pages 270–274, 1978. 75

List of Figures

List of Tables

List of Abbreviations

AOI ..... Area of Interest
API ..... Application Programming Interface
AR ..... Augmented Reality
CAVE ..... CAVE Automated Virtual Environment
CCD ..... Cyclic Coordinate Descent
DAE ..... Digital Asset Exchange
DCC ..... Digital Content Creation
DLP ..... Digital Light Processing
DOM ..... Document Object Model
FIFO ..... First-In First-Out
GUI ..... Graphical User Interface
HCI ..... Human Computer Interaction
HMD ..... Head Mounted Display
HUD ..... Head Up Display
IK ..... Inverse Kinematics
ISS ..... International Space Station
JKU ..... Johannes Kepler University
IDE ..... Integrated Development Environment
JT ..... Jacobian Transpose
LERP ..... Linear Interpolation
LGPL ..... GNU Lesser General Public License
MR ..... Mixed Reality
MVC ..... Model View Controller
NVE ..... Networked Virtual Environment
ODE ..... Open Dynamics Engine
p2p ..... peer-to-peer
SLERP ..... Spherical Linear Interpolation
STL ..... Standard Template Library
TCP ..... Transmission Control Protocol
UDP ..... User Datagram Protocol
UML ..... Unified Modelling Language
USB ..... Universal Serial Bus
VE ..... Virtual Environment

VR . . . . . Virtual Reality
VRC . . . . . Virtual Reality Center
XML . . . . . Extensible Markup Language