Augmented & Virtual Reality:  
Development of applications for a usability study

Written by Frank van Huussen, Pauline van der Horst and Jan-Willem Baggerman

April – June, 2006

Supervised by:  Paulo Dias (IEETA) and  
Rafael Bidarra (EEMC)

BSc coordinator:  Matthijs Sepers (EEMC)

Sending university:  Electrical Engineering, Mathematics and Computer Science (EEMC)  
Delft University of Technology (TU Delft), The Netherlands

Receiving university:  Institute of Electronics and Telematics Engineering of Aveiro (IEETA)  
University of Aveiro (UA), Portugal
Preface

This report concludes our internship at the University of Aveiro, which was made possible thanks to the Erasmus Exchange Program between this university and TU Delft.

We would like to thank our advisors from TU Delft, dr. ir. R. Bidarra and ir. M. Sepers, for giving us the opportunity to work at this project and for their support from The Netherlands during our stay of ten weeks in Portugal. We would also like to thank our advisor at UA, P. Dias and PhD student A. Pimentel, not only for their advice and support during the project, but also for taking the time to make us acquainted with Portugal. At last, we like to thank J. Madeira and B. Sousa Santos from UA to be there for us when advises were needed.

Aveiro, Portugal, June 2006

Frank van Huussen,
Pauline van der Horst,
Jan-Willem Baggerman
# Table of contents

1. Introduction ........................................................................................................... 1  
2. Objectives ............................................................................................................... 2  
   2.1 Augmented Reality objective ........................................................................... 2  
   2.2 Virtual Reality objective .................................................................................. 2  
3. Organization ........................................................................................................... 3  
   3.1. Communication ............................................................................................... 3  
   3.2. General planning ............................................................................................ 3  
   3.2. Planning of the AR application ........................................................................ 4  
   3.2. Planning of the VR application ........................................................................ 4  
4. Augmented and Virtual Reality .............................................................................. 5  
   4.1 Augmented Reality ........................................................................................... 5  
   4.2 Virtual Reality ................................................................................................ 5  
   4.3 Similarities ........................................................................................................ 5  
5. Software .................................................................................................................. 6  
   5.1 Microsoft Visual Studio 2003: C++ .................................................................... 6  
   5.2 ARToolkit ......................................................................................................... 6  
   5.3 The Visualization Toolkit: VTK .......................................................................... 6  
   5.4 OGRE ................................................................................................................ 7  
   5.5 GtkRadiant ......................................................................................................... 7  
6. Hardware .................................................................................................................. 8  
   6.1 PC ....................................................................................................................... 8  
   6.2 HMD .................................................................................................................. 8  
   6.3 Webcam ............................................................................................................. 8  
   6.4 Wireless camera ............................................................................................... 8  
   6.5 Tracker .............................................................................................................. 9  
   6.6 Accelerometer ................................................................................................... 9  
7. Design ....................................................................................................................... 10  
   7.1 Design of Augmented Reality application ......................................................... 10  
      UML diagram ...................................................................................................... 10  
   7.2 Design of Virtual Reality application ................................................................. 11  
      7.2.1. Development .......................................................................................... 11  
      7.2.2. Gameplay ............................................................................................... 11  
      7.2.3. Control ................................................................................................. 11  
      7.2.4. UML diagram ....................................................................................... 12  
      7.2.5. Class explanation: the OGRE engine ......................................................... 12  
      7.2.6. Class explanation: the Maze game ........................................................... 13  
8. Implementation ....................................................................................................... 15  
   8.1 Implementation of the AR application ................................................................. 15  
      8.1.1. Program structure .................................................................................. 15  
      8.1.2. General principles ............................................................................... 16  
      8.1.3. Models ................................................................................................ 21  
   8.2. Implementation of the VR application ............................................................... 27  
      8.2.1 Loading a BSP level ................................................................................. 27  
      8.2.2 Collision detection ................................................................................. 27  
      8.2.3 The Player object .................................................................................... 31  
      8.2.4 The targets .............................................................................................. 33  
      8.2.5 Controlling the game .............................................................................. 34  
      8.2.6 Configuration ......................................................................................... 37  
      8.2.7 Multiple program states ....................................................................... 38  
9. Experiments ............................................................................................................ 40
1. Introduction

This report presents the development of an Augmented Reality (AR) application to visualize research data and the development of a 3D video game. These applications are completely developed at the University of Aveiro (UA) as a part of the Bachelor project IN3700 of TU Delft. The AR application is used for a qualitative study which visualization method is most natural for human perception. The 3D game is used for a quantitative study of usability using different navigation methods.

Here follows an description of the contents of each chapter; our objectives are covered in chapter 2. How we organized this projects is discussed in chapter 3. Next, the definitions of Augmented and Virtual Reality will be described and in chapter 5 and 6 covers the software and hardware we used. Chapter 8 will cover our implementation and then a the usability experiments will be described. Chapter 10 coves our technical conclusions and the last chapter are the references to literature. The three appendices will cover the class diagrams, setup of the software, an example of the output of the maze game, a description of changed sourcecode within the OGRE framework and at last the recommendations.
2. Objectives

In contrast to traditional desktops, there has been done little research to validate the utility and usability of Virtual and Augmented Reality environments. The evaluation of usability of these new technologies is very important to design systems that are more intuitive than a traditional desktop. Such an evaluation is also important for future development of applications that can gain from these two new technologies. Therefore at IEETA, a research departments of the UA, there is running a qualitative and quantitative study of Augmented and Virtual Reality environments. To make such a study possible, we developed both an Augmented Reality and a Virtual Reality application.

2.1 Augmented Reality objective

Visualization of data is used at large scale, for example a medical surgeon can use a traditional desktop computer to study a model of a human lung, before he has to perform the surgery. This leads to an interesting study if an alternative display method is more natural for human perception.

Models of research data were usually made in VRML, which is a language in which 3D objects can be described. Because some limitations of VRML, like the lack of volume rendering, a software library called Visualization ToolKit [VTK] became popular, especially for modelers of construction engineering and medical data. VTK offers hundreds of classes for the construction, manipulation and visualization of 3D data. This makes VTK very useful for creating and displaying complex models.

The manipulation of research data is usually done with a traditional desktop computer, with interaction from a mouse and a keyboard. When studying alternative display methods, the researchers at IEETA make use of Augmented Reality (AR). Visualization of computer generated 3D models in the real world can be done with a software library called Augmented Reality Toolkit [ARToolkit]. Unfortunately, this toolkit has two major drawbacks. First ARToolkit is only capable of reading VRML-models and when these models are too complicated, they can’t be handled correctly anymore. Secondly, ARToolkit doesn’t support stereoscopic vision, which is an important point of interest for this study.

Therefore it is important to combine ARToolkit with VTK for future research. Such a configuration provides much more flexibility for displaying research data and studying which display method is the most natural for human perception.

2.2 Virtual Reality objective

To compare the difference between the usability of a Virtual Reality environment and a normal desktop environment, the idea is to have a game developed especially for this research. This game is supposed to be controlled in two ways, by a VR environment and by a desktop environment. When many users will play the game in these two different environments, their performance can be compared. It is expected there will be a noticeable difference in the performance when playing exactly the same game in a different environment.

To be able to compare the performance of a user in the game, it is needed the game provides the researchers with statistical data. These data will be stored in a file which later can be analyzed to collect the data of all users together. The application will save all events happening during the gameplay, and the time the event happened. It is recorded how well the user can navigate within the environment and how fast the user can complete an objective. These data will provide the researchers at IEETA with enough information to decide if a user usually performs better in a Virtual Reality environment or a normal desktop environment.
3. Organization

This chapter covers the organization of our two projects. In the first paragraph the communication between our team and our supervisors. Next the general planning is discussed, and then the specific planning for the two different applications will be covered.

3.1. Communication

After we arrived, the first we did was meeting our advisors. Our head adviser is Paulo Dias, which we had to go to with problems or questions. To assist Paulo Dias, we also can turn to the two co-advisors Joaquim Madeira and Beatriz Sousa Santos. During the project we would cooperate with PhD student Angela Pimentel, who would write her final essay about the usability of our applications. With this team we had weekly meetings to report our progress. Besides, because the usability experiments weren’t completely structured yet, we discussed possibilities in which way our application could further be developed.

3.2. General planning

In the first week we would make ourselves familiar with the software and hardware that we would need during the project. This includes:

- Studying the Magic Book examples to make ourselves familiar with ARToolkit;
- Studying applications that make use of the orientation tracker and accelerometer;
- Connecting the Head Mounted Display and try it with different applications;
- To notice the difference of stereoscopic vision and checking compatible environments;
- Study the references and documentation about ARToolkit and VTK

When we had a basic idea of what we could expect, the following general planning was constructed:

<table>
<thead>
<tr>
<th>Week 1</th>
<th>Make ourselves familiar with the software and hardware, which we would need</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 2</td>
<td>Make a rough layout for the game, start the Magic Book application</td>
</tr>
<tr>
<td>Week 3</td>
<td>Create the collision detection engine and a simple level for the game, finish the Magic Book application</td>
</tr>
<tr>
<td>Week 4</td>
<td>Finish the level, make user input work correctly and report collisions in a logfile.</td>
</tr>
<tr>
<td>Week 5</td>
<td>First deliverable version, first tests with the game. First iteration in the development process: improve the game at desired points.</td>
</tr>
<tr>
<td>Week 6</td>
<td>Next iteration if needed. Add static objects that have to be found and dynamic objects that have to be avoided and add a GUI and sound engine if time will let us</td>
</tr>
<tr>
<td>Week 7</td>
<td>Next iteration if needed. Write the final report. If time will let us, add more options, levels, objectives, models, and so on.</td>
</tr>
</tbody>
</table>

It took a long time to work through the first stage, because we encountered a lot of problems to get the existing applications running. Especially the settings of the right includes and libraries was a huge problem. Besides, the wrong drivers for the video card was installed, so stereoscopic vision didn’t work and the orientation sensor wasn’t responding as expected, but this seems to be an already known problem. After solving these problems, we could start with the rest of our planned tasks.

In the second week we split our group into two subgroups. Jan-Willem would already start to make the framework of the game, while Frank and Pauline start to develop the AR application. Soon we realized that the AR would also be a large project on its own and would also take seven weeks for development. Therefore Jan-Willem continued with the development of the game and Frank and Pauline continued the development of the AR application. Because we now had to make two different versions of the planning, both are included in this chapter.
3.2. Planning of the AR application

Because such a project never has been done before, it was hard to construct a good planning. Therefore we came up with a global plan, including steps which would take us to the final application:

<table>
<thead>
<tr>
<th>Week 2</th>
<th>Grabbing the video stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 3</td>
<td>Attach a fixed model to the detected marker</td>
</tr>
<tr>
<td>Week 4</td>
<td>Transform the simple virtual model, as it was attached to the marker</td>
</tr>
<tr>
<td>Week 5</td>
<td>Implement the more complex models</td>
</tr>
<tr>
<td>Week 6</td>
<td>Finalize the source code and write the final report</td>
</tr>
</tbody>
</table>

3.2. Planning of the VR application

Making a game on your own is something completely new. In a previous project in the Netherlands, a game was created with a group of five persons. Planning how long everything would take on your own was a difficult task, and eventually this scheme proved itself wrong, due to some unexpected problems when working on the collision detection.

<table>
<thead>
<tr>
<th>Week 2</th>
<th>Make a rough layout for the game, begin working at the collision detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 3</td>
<td>Finish the collision detection engine and make a simple level for the game</td>
</tr>
<tr>
<td>Week 4</td>
<td>Finish the level, make user input work correctly and report collisions in a logfile.</td>
</tr>
<tr>
<td>Week 5</td>
<td>First deliverable version, first tests with the game. First iteration in the development process: improve the game at desired points.</td>
</tr>
<tr>
<td>Week 6</td>
<td>Next iteration if needed. Add static objects that have to be found and dynamic objects that have to be avoided</td>
</tr>
<tr>
<td>Week 7</td>
<td>Next iteration if needed. Add a GUI, and maybe a sound engine.</td>
</tr>
<tr>
<td>Week 8</td>
<td>Next iteration if needed. Write the final report. If there might be enough time, improve game and add more options, levels, objectives, models, and so on.</td>
</tr>
</tbody>
</table>
4. Augmented and Virtual Reality

This chapter covers the differences and similarities between Augmented Reality and Virtual Reality and defines how these two subjects are defined in this report.

4.1 Augmented Reality

Augmented Reality (AR) is a field of computer research, where live images of the normal physical world are combined with computer generated models. AR makes it possible for a user to see these virtual models as a part of the real world. The real world can be enriched with extra information. For example, with AR a better understanding of relative distances between the virtual model and the objects in the real world can be achieved.

4.2 Virtual Reality

In contrast to AR, in Virtual Reality (VR) everything you see is computer generated. This means that not just the models, but also the world itself only exists as a virtual model.

An example for a VR environment is architectural software. When an architect designs a building, usually one of the first prototypes will be displayed as a computer generated model. This is an aid for the architect to show potential buyers how the finished building should look like.

Another example of a VR environment which is very common, is computer gaming. A computer game consists of a virtual world with virtual objects. This gives game designers a great level of freedom to design a game with a different reality that does not exist in the real world, but only in fantasy. When somebody moves inside a game however, he will learn this environment and get to know hallways and rooms like it exists for real.

4.3 Similarities

In both environments the user experiences that all objects have a position which have a relative distance to his body. In this way the object moves in proportion to the movements of the user’s body. In both environments the virtual part is interactive. The user requires real time response from the system so he will be able to interact with it the environment in an effective manner.

Both environments can be displayed on a 2D screen (on a normal desktop computer) or with a display technology like a Head Mounted Display (HMD). That means that we can create four different environments:

- Augmented Desktop (AD)
- Augmented Reality (AR)
- Virtual Desktop (VD)
- Virtual Reality (VR)

When using a HMD display technology for the AR and VR environments, the user can obtain 3D stereoscopic vision. In the AR case, only the virtual models are 3D and in the VR case the entire world and all of its virtual models are in 3D. Using a 3D stereoscopic vision will help giving the user a better understanding of how the virtual model or virtual world should look like if it existed in the real world.
5. Software

When developing our applications, we used different software and libraries. This chapter provides more detailed information about the software used in our project.

5.1 Microsoft Visual Studio 2003: C++

For this project, we implemented our applications in the C++ language. One of the most used development environments for C++ projects is MS Visual Studio. This environment helps developers with the IntelliSense tool, which makes browsing through huge numbers of classes a lot easier by showing which methods could be called inside an object. Visual Studio also keeps track of intermediate compilations of a program, so each time something has been edited and has to be compiled again, only the edited classes need to be recompiled and compile time is greatly reduced.

5.2 ARToolkit

ARToolkit is a C language software library for building Augmented Reality applications. This toolkit is capable of detecting the orientation of special markers in the real world. This way it is possible to render virtual models according to the orientation and position of the detected markers in the real world.

The most difficult part of creating an Augmented Reality application is to track the user’s viewpoint. ARToolkit uses computer vision techniques to process each captured video frame of the real world to search for any squared shapes. If a thick black border is found, the orientation and position of this potential marker are calculated. If this figure is really detected as the predefined marker, ARToolkit will transform the virtual model to place it on top of the marker. The next diagram summarizes the steps above:

![Figure 5.2: ARToolkit diagram (figure from http://www.hitl.washington.edu/artoolkit/documentation/userarwork.htm)](http://www.hitl.washington.edu/artoolkit/documentation/userarwork.htm)

5.3 The Visualization Toolkit: VTK

VTK is an open-source, freely available cross-platform visualization library written in C++. It is used for computer graphics, visualization and image processing. VTK provides a large variety of data representations and readers and writers to exchange data with other applications. VTK’s open-source code enables to possibility to easily extend it for developing a new application. Every VTK program contains at least the following minimal pipeline, but extensions of this pipeline are possible:
1. First the data must be read from a file or created from scratch to construct a member of `vtkSource`;

2. The data needs to be mapped with `vtkMapper` to move the data from VTK to the graphics primitive, like OpenGL;

3. For setting properties to the data, such as color and position, the data needs to be converted to a `vtkActor`;

4. `vtkRenderer` defines a rectangle on the computer screen in which VTK can render the images;

5. `vtkRenderWindow` takes care of adding a window to the renderer. For example, a title bar and a closing button will be added.

### 5.4 OGRE

For the game used in the Virtual Reality experiment, the graphical engine OGRE [OGRE] is used. At the moment of development in this project, version OGRE (Object-Oriented Graphics Rendering Engine) provides developers with an intuitive class library to produce applications with 3D graphics, without having to understand every detail of the rendering itself. The OGRE structure is modular and every feature is clearly documented and separated in the class hierarchy. Another advantage of OGRE is the fact the engine is open-source, so this makes it easy to change parts of the engine, in case some feature has not been implemented yet or has to be changed to fit the application’s. The OGRE SDK comes with a lot of sample applications, which demonstrate a lot of functions and application structures.

### 5.5 GtkRadiant

The virtual environment for a game can be created in GtkRadiant [GtkRadiant], which is an open source level editor. GtkRadiant is based on QERadiant and Q3Radiant, level editors used by ID Software for developing levels for Quake II and Quake III Arena. Levels in GtkRadiant are saved in the BSP (Binary Space Partition) format, a format commonly used in computer games like Doom and Quake. Unlike other level editors, the BSP exporter is integrated in GtkRadiant. Creating levels in GtkRadiant is done by creating boxes (called brushes), and then cutting away the pieces not needed.

![Figure 5.3: The VTK pipeline](image1)

![Figure 5.4: Design of a game level in GtkRadiant](image2)
6. Hardware

Extra hardware is required when using Augmented or Virtual Reality. This chapter presents all the hardware that was used in both environments.

6.1 PC

All the software will run on a personal computer that is capable of controlling all other hardware used by the applications. The computer is equipped with an NVIDIA GeForce FX 5950 Ultra video card [NVIDIA], capable of providing 3D DDC line signals for the head mounted display. It has a fast 2.5 GHz processor and 1024 MB of RAM memory, ensuring the system will not run out of resources during the running of the AR and VR applications.

6.2 HMD

A Head Mounted Display, or HMD, is a display device that can be put on the head, just like normal glasses. However, the glasses consist of two small displays which can be controlled individually. By feeding each eye with a slightly different image, the person wearing the HMD sees a 3D image and experiences the virtual environment at a more natural way (as if he is standing there for real).

The HMD used is an i-glasses Video 3d pro display manufactured by iO Display Systems [iGlasses]. This display has a 800x600 pixels resolution, providing a sharp and clear image. The virtual display renders a size of a 70 inch monitor watched at a distance of 4 meters. The maximum input refresh rate is 100 Hz. When in 3D mode, the HMD will quickly swap sequential frames to both eyes, this means there will be a maximum rate of 50 Hz for each eye.

6.3 Webcam

A webcam is capable of capturing images from the real world and send these images to the computer. The Creative Webcam NX Pro [Creative] is a such a camera and has a CMOS sensor with a resolution of 640x480 pixels. This device must be connected to the computer with USB.

6.4 Wireless camera

When using a HMD, a camera must be used to obtain video see-through. This camera is mounted directly at eye-level in front of the HMD, so the captured images can be displayed in front of the user’s eyes. To mount a camera on the HMD, its dimensions must be small. Therefore a wireless analog micro camera is used. This camera has a CMOS sensor with a resolution of 330 lines, transmitting a PAL color signal at 1.2GHz. To power this camera it must be connected to an adapter or a 9V battery.

The receiver can be adjusted manually fine-tuned to obtain the best reception of the analog 1.2 GHz signal. Its output is composite PAL color signal and must be connected with a cable with RCA jack plugs.

To convert this signal to USB, a Real TV box top USB manufactured by NPG Technology [NPG] is used. It is connected with USB 2.0 to the computer and outputs a PAL signal with 25 frames per second by a resolution of 768x576 pixels.

Figure 6.1: the i-glasses HMD
Figure 6.2: Wireless camera
Figure 6.3: Video receiver
Figure 6.4: Real TV box
6.5 Tracker
An orientation sensor will be mounted on the HMD to keep track of the position of the users head. This is a motion tracker from InterSense [InterSense], the InterTrax 2. This sensor does an angular tracking in three degrees of freedom (roll, pitch and yaw) and is very fast, with an update rate of 256 Hz and an internal latency of only 4 ms. This makes the sensor ideal for real time VR applications.

6.6 Accelerometer
When walking through the VR environment, the user can control the application with an inertial sensor he is holding in his hand. This accelerometer gives an RS232 output and is based on the LIS3L02AQ chip from STMicroelectronics [STMicro] which can detect accelerations in three directions (x, y and z axis).
7. Design

Because the two developed applications cannot be designed by the same approach, this chapter describes the design stage of the two applications separately.

7.1 Design of Augmented Reality application

We found it necessary to use a software process model to determine the order of the stages involved in our software development and evolution. Secondly we want to establish the transition criteria for progressing from one stage to the next. These include completion criteria for the current stage plus choice criteria and entrance criteria for the next stage.

Because neither of us had ever worked with ARToolkit or VTK before, we started with doing research on both toolkits. After reading documentation about these subjects, having a look at their references and studied the delivered examples of both toolkits, we started to get an idea of their basic structure.

One of the first things we discovered that might lead to problems was that ARToolkit was written in C and VTK in C++. We found out that it is possible to use C code in a C++ application, but not the other way around. Therefore we came up with the following general plan on how to proceed to the implementation stage: we will base our application on VTK and we will implement the required functions of ARToolkit. Then we will grab the video stream from ARToolkit and render this on a VTK window. Our next goal would be to move a simple VTK object according to the position and orientation of the marker. When this would work properly with a simple model we could start with more complicated models.

The application that we are going to develop, is never build before. Therefore there was no documentation, examples or field experts available that could help us specify our global model. When starting with this project, we didn’t even know if our goal would be achieved using the general plan described above. That’s why we decided to work according to the transform model:

The transform model assumes the existence of a capability to automatically convert a formal specification of a software product into a program satisfying the specification. The steps prescribed by the transform model we followed are:

- We made a formal specification of the best initial understanding of the desired product.

- We transformed each of the described steps of the specification into code. These steps included: grab a video stream from ARToolkit and render this on a VTK window, move a simple VTK object according to the position and orientation of the marker, use more complicated models.

- Then we exercised the resulting model with the video stream and webcam. This included checking if the model transformed and scaled correctly. Checking, when present, if the interaction worked well. If this was the case the model was tested by external testing persons in different environments.

- The next step was to make adjustments based on the resulting test experience, and to optimize, and again exercise the adjusted models.

**UML diagram**

In all our models a basic structure will be present. This structure begins with passing video stream information from ARToolkit to VTK and ends with following the pipeline of VTK. Therefore all the models will have included the following variables and methods:
Because the control takes place from one basic class and therefore the structure is usually expressed as a simple hierarchy. The more complicated models we have to implement will be controlled from the main basic class. The control in the program will always follows a fixed pattern. In this way there is a chain of processes arranged so that the output of each element of the chain is the input of the next beginning and ending in the main class. This pipeline and hierarchy structure reduces complexity and makes it easier to implement a new model. To understand this structure in more detail, the UML diagrams of four the models (see Appendix A) will show that the control is indeed controlled by the one basic class.

### 7.2 Design of Virtual Reality application

The VR Application will be a simple computer game, which can be controlled in two ways: a desktop environment and a virtual environment with HMD and tracker device. This game has to provide the researchers at IEETA with all data they need to compare a users behavior in a desktop environment and a VR environment.

#### 7.2.1. Development

Because this project has to be done in a very limited time, the game will be developed in the beginning as simple as possible, adding further functionality later. Most of all, it is important to have a stable, robust and effective application which generates the variables needed for comparing the performance of the user in a desktop environment and the user in a virtual environment. The application will first be mainly focused on navigating through the environment, controlling the game with two navigation modes (virtual and desktop) and generating any variables out of the navigation part. When the simple application works, this can be extended later to have more functionality and objectives for the user. This approach makes an iterative development process the right choice for developing this application, where an iteration might be optional eventually, since the limited time might not allow for multiple deliverable versions of the system. The approach during development will be partly bottom-up since the OGRE framework is providing various detailed modules, and partly top-down since first the rough application has to be thought about, before further concentrating on details (like a sound engine).

#### 7.2.2. Gameplay

The game will be a ‘maze game’; a game consisting of a labyrinth, where the user will have to navigate through. In this maze, objects are placed through the entire level. The objective for the user is to collect as many objects as possible, before he is out of time. The objects are equally spaced through the maze, forcing the user to find his way through the entire level before he is able to finish the game. Because the game has to provide data about the users behaviour, the game will count the number of collisions with walls, the number of objects found, the total distance travelled and the time spent playing the game.

#### 7.2.3. Control

In the desktop environment, the user can control the movements inside the game with the keyboard and look around by moving the mouse, a way of control similar to most computer games. When the user is navigating in the VR environment, the position of his head will be passed to the game by the tracker, which makes looking around feel natural. The original idea was to control the movements
inside the game with the accelerometer, but this has been changed into using the mousebuttons to walk forward and backward. More about this can be said in the Implementation chapter.

7.2.4. UML diagram
For this game the Ogre framework will be used. This will make developing the game a lot easier, since Ogre will handle all graphic functions, and only the logic of the game has to be implemented. Because OGRE is not a game engine however, game-specific features like collision detection and sound is not directly available within OGRE. It is possible to integrate all kind of libraries for these functions when needed, or implement other functions from scratch.

Here you can see the UML diagram of the game, containing a number of objects derived from Ogre. This diagram is kept small, just to give an idea of the structure of the program. For more detailed class definitions, see Appendix A.

Figure 7.2: the UML diagram of the game. The grey classes are part of the OGRE Engine

7.2.5. Class explanation: the OGRE engine
The OGRE Engine consists of many classes and objects, the main ones are shown in the class diagram in figure 7.2. To have a better idea of the framework that is used in this project, these classes will be shortly commented, following the OGRE manual. [OGREman]

Root
The most important object is the Root object. This is the entry point to the OGRE system. The Root class has various methods to configure the system, methods to obtain pointers to other objects in the system, such as the SceneManager, RenderSystem and various other resource managers. Also the Root class has a method to enter the continuous rendering loop, needed while playing the game.

RenderSystem
The RenderSystem class is defining the interface from OGRE applications to the 3D API. This class is an abstract class, able to interface with different types of 3D APIs (like Direct3D or OpenGL). After
the system has been initialised, the RenderSystem object is available through the Root object. Most applications will however not need to access this RenderSystem directly, since the SceneManager will take care of most rendering commands. So the RenderSystem exists, but does its work silently inside the OGRE framework.

SceneManager and SceneManager Enumerator
There are many different types of scenes in different types of games. For rendering an indoor scene, for example, you will need different algorithms than for rendering an outdoor scene. A flight simulator game (with most objects far away) will also need different rendering algorithms than a car racing game (most objects close to the user). This is where the use of a SceneManager comes in: a SceneManager is organising all elements a game would consist of, deciding which objects have to be rendered for the situation given. For indoor scenes, there is a BSP SceneManager (using the Binary Space Partition format, also used in this project). For outdoor scenes, there are other SceneManagers, for example a Terrain SceneManager (which uses a heightmap, terrain texture and detail texture). The SceneManager Enumerator is a class that simply controls what SceneManager can be used for which scene type.

Material and TextureLayer
When an object has to be rendered, the SceneManager will need the objects geometry (defined in the Mesh object) and the material that has to be put on the object. This is defined in the Material and TextureLayer objects. The TextureLayer object contains the texture that has to be put on the object, but an object with only a static texture would not be really appealing for the eye. This is why OGRE has a Material class, which enables a developer to combine different textures, enter properties like shininess, reflectance, what lighting and filtering modes should be enabled, and so on.

Entity and Mesh
As stated in the part about the Material and TextureLayer classes, a game object also need to have a definition of its geometry. The Entity object is an instance of a game object, which can be any movable object inside the game. The Mesh object is the definition of the geometry of the game entity, loaded from a .mesh file. An Entity can have only one Mesh, but the same Mesh can of course be used for different game entities. An Entity however is not enough to display the game object, the SceneManager also needs to know what location the game object has. This is done by the SceneNode class.

SceneNode
OGRE uses a hierarchical structure to keep track of all game objects. This structure consists of a lot of movable SceneNodes, to which all kind of Entity objects can be attached. Each SceneNode can have other SceneNodes as children, creating a tree-like structure. The location of each SceneNode is relative to each other, so moving one SceneNode will also move all of its children. This is a great structure to easily keep track of and modify the location of each object inside the game.

Camera
The last object that is derived from the OGRE engine, is the Camera object. A Camera defines where a viewpoint is that the RenderSystem will use. In a game, multiple Camera objects might be possible if needed. This object behaves like any Entity and also needs to be attached to a SceneNode to become part of the scene of the game.

7.2.6. Class explanation: the Maze game
The Maze game consists of a couple of classes, some of them derived from existing classes that were provided as sample applications for the OGRE framework. The main purpose of each class will be explained here.

MazeApplication
The MazeApplication class is the main class that will be started when the game is loaded. In this class, all settings are loaded and all essential objects are created. After all resources are loaded (level,
models, textures) while displaying a progress indicator, the scene will be created and the game will start listening to the user input by creating a MazeFrameListener. Eventually, this class is also capable of displaying interfaces through CEGUI, a more advanced user interface within OGRE, but that functionality is left out in this version for now.

**MazeLoadingBar**
The MazeLoadingBar class is totally derived from an OGRE Reference application. This class displays a loading bar while loading all resources, and is created and destroyed by the MazeApplication. The intention is to edit this progress bar later to a more custom design and code, but at this moment the bar is still exactly the same as in the original Reference application.

**MazeFrameListener and SerialSensor**
The most important class in the game is the MazeFrameListener, containing most of the game logic. In this class, all user input will be handled and evaluated. After this, the MazeFrameListener can decide if a certain player movement is allowed and which objects have to be moved inside the game. The MazeFrameListener is also in control of putting the game in different ‘game states’, enabling the application to halt a moment, for example to show the user a message when he ran out of time. Because the user input should not only come from the keyboard and mouse, the MazeFrameListener also needs a way to communicate with the tracker and accelerometer. The tracker has a dll file that can be linked, the SerialSensor class takes care of reading out the accelerometer.

**ODE**
In OGRE, there is no built-in solution for collision detection. This can be handled through external collision or physics libraries, such as Opcode (a plain collision detection library) or ODE (a little too advanced physics library). The implementation of Opcode should be more easy. However, the OGRE SDK provides a Reference application which already uses ODE, so this was a nice starting point with an example how to implement collision detection (and physics) in the game. Also, after several tests ODE gave better results than Opcode. Eventually, only very little functionality of ODE is used, the physics engine is available but not used yet.

**ObjectManager and ApplicationObject**
The ODE implementation in the Reference application of the OGRE SDK uses a World class and an ApplicationObject class. The World class incorporates simulation settings and object manipulation to mimic the physics of a real world, but because this part will not be used, the World object is renamed to ObjectManager in this application. The ObjectManager class takes care of the creation and tracking of collisions of objects. All objects that should be able to track collisions or have physics, are extending the ApplicationObject class. This class is passing the movement and physical properties of an object to the ObjectManager.

**Obj_Player**
The player will be represented by the Obj_Player object, so movements of the player can be affecting the collision detection. This object is extending the ApplicationObject class, defining what exact physical parameters the player has. Also, because a player can fall (or jump, optionally), the player object has some methods for this. Finally, the Obj_Player will keep track of the total travelled distance, the total number of targets found, and the writing of the logfile at the end of the game.

**Obj_Box**
The Obj_Box object is also extending the ApplicationObject class, like the Obj_Player. This object however represents the targets the player has to find. Because both the player and the targets are extending the ApplicationObject class, a target can be alerted by the ObjectManager when a player is colliding with it. When alerted, the target disappears and will play a short sound to let the user know he found the target.
8. Implementation

This chapter will describe the implementation details of both applications separately.

8.1 Implementation of the AR application

This paragraph will cover three main subjects, first the general program structure of all four applications will be described, next the general principles which were used and last the specific details about the different implemented models.

8.1.1. Program structure

The final project deliverables includes four different datasets, which have been visualizes in an Augmented Reality environment. However, all of these four different applications are based on the same basic application with the same structure. The whole program exists of six functions: main, init, vtkInit, arVideoCapStart, argMainLoop and vtkRender. The application and use of these methods will be discussed here.

```c
int main(int argc, char **argv)
```

The main function calls the init, vtkInit, arVideoCapStart and argMainLoop successively.

```c
static void init(void)
```

This function is needed to initialize ARToolkit routines for starting the video capture, reading of the marker and camera parameters, and the setup of the graphics window. At first, the video path is opened and the video image size is found. Then the camera parameters are set and printed to the screen. Finally a graphics window is opened, which is needed for ARToolkit to function properly, but is not used in our application.

```c
static void vtkInit(void)
```

In this function the basic VTK pipeline is created. First actors must be created, so to import the images from the webcam, an data object called vtkImageData needs to be created. This data object represents a geometric structure that is a regular array of points used to create an image, with all the needed properties like number of colors and the dimensions. Next a vtkImageData is constructed with as input the data from vtkImageData. Now a vtkActor is created, it can be placed in the scene. First the bottom left point of the image is placed at the origin of the scene. Its dimensions are the same as the chosen webcam resolution and is oriented in the (x, y) plane. Also the models are an vtkActor and therefore these needs to be constructed too. Most models are read from a file and were processed using different methods. Details of this construction can be found later in this chapter.

When actors are created they need to be rendered. Rendering is the process of converting geometry, a specification for lights, and a camera view into an image. vtkRenderer performs coordinate transformation between world coordinates, view coordinates (the computer graphics rendering coordinate system), and display coordinates (the actual screen coordinates on the display device).

To get an actor rendered it must be add to the vtkRenderer. The whole scene settings must be set in a render window, called vtkRenderWindow. A render window is a window in a graphical user interface where the renderers draw their images. As needed with the models, this window allows rendering in stereo vision.

```c
arVideoCapStart();
```

This function call of ARToolkit start the capturing of video images from the webcam.

```c
argMainLoop(NULL, keyEvent, mainLoop);
```

This is the routine where the bulk of the ARToolKit function calls are made. First a video frame is captured using arVideoGetImage. Then the function arDetectMarker is used to search the video image
for squares that have the correct marker patterns. The number of markers found is contained in the variable `marker_num`, while `marker_info` is a pointer to a list of marker structures containing the coordinate information and recognition confidence values and object id numbers for each of the markers. Next the video frame is unraveled in bytes for each pixel, processed and saved. The video image is now grabbed and has served its purpose, so it isn’t needed anymore: the frame grabber can start grabbing a new frame. Therefore `arVideoCapNext` is called again. Next, all the confidence values of the detected markers are compared to associate the correct marker id number with the highest confidence value. If no patterns are found (k == -1) a boolean called `markerFound` is set to false and then it will jump back to the beginning of `argMainLoop`, else this boolean is set to true.

```c
static void vtkInit(void)
```

All the preprocessed data is now collected and `vtkRender` is called to update the actors with the new information and render the newly obtained scene. The transformation values retrieved in `argMainLoop` are used to transform the actors of the model. If a marker is found in `argMainLoop` the model is set visible, else only the background video images will be shown. At this moment all the data and information for the new frame setting is ready and everything is rendered, this the control is given back to the `argMainLoop`.

### 8.1.2. General principles

The four different applications of the four models are all based on the same underlying principles, but with different parameters. First the general principles will be discussed and then the application of these principles to each model.

**Reading image**

ARToolkit extracts an image from the webcam and stores this in the computer memory (in BGRA format: blue, green, red and alpha channel). This is done by the function `arVideoGetImage`. This function also returns a pointer to the first pixel. Each pixel consists of four values (BGRA) which all confiscate one byte. Therefore, for reading an image, it’s needed to read width * height (resolution of the webcam) * 4 bytes (1 byte for each color and alpha value) from the memory. While reading this image from the memory each byte is put in a successive element of an array called `pixels`:

```c
// Grab BGRA data from memory and store it in the "pixels" array
for (int i = 0; i < (xsize * ysize * 4); i++) {
    pixels[i] = *(dataPtr + i);
}
```

Because ARToolkit provides the image in BGRA format and VTK uses the RGB format, a conversion must be constructed. Therefore, after reading and storing an image in an array, its pixel orders needs to rearrange and the A (alpha) values needs to be removed:

```c
// Reorder pixels from BGRA to RGB and store in the "pixelsRGB" array
for (int i = 0; i < (xsize * ysize); i++) {
    pixelsRGB[i * 3 + 0] = pixels[(i * 4 + 2)];
    pixelsRGB[i * 3 + 1] = pixels[(i * 4 + 1)];
    pixelsRGB[i * 3 + 2] = pixels[(i * 4 + 0)];
}
```

When converted this image data to an array with RGB format, VTK needs to know where to find this image data. VTK has a function `GetScalarPointer` which retrieves a pointer to the memory location where is expects to find the first pixel data of the image. Therefore when putting the image data from the `pixelsRGB` array in successive memory locations behind this pointer location, VTK reads the image automatically from this place. Because this process has to be repeated for each image the pointer each time has to be set to its native point of the scalar data. This is done by the function `AllocateScalars`.
```c
data = (unsigned char*) imageData->GetScalarPointer();
imageData->AllocateScalars();
for (int i = 0; i < (xsize * ysize * 3); i++)
    *(data + i) = pixelsRGB[i];
```

The process results in a smooth video stream in a VTK render window.

**Positioning the object onto the marker**

Having a smooth video stream, a simple cone model can be put in the middle of the marker. The most obvious start was to fix the cone in the origin of the 3D axis and transform the virtual camera of VTK according to the movements of the webcam. This is the same approach ARToolkit is using in its own application. We did a lot of research on ARToolkit’s transformation matrix to transform the virtual camera of VTK. After many failed attempts we tried a different approach: to transform the model according to the camera view and then fix this model on the marker. Therefore information about the marker is needed. The variable `marker_info` provided by ARToolkit contains the x and y position of the centre of the marker. These two coordinates can be used to position the origin of the VTK model in the middle of the marker:

```
// Set the cone at the place of the marker
coneActor->SetPosition(marker_info->pos[0], marker_info->pos[1], 50);
```

Often the origin of the model isn’t the same as the center of the model, or another part of the model must be set in the middle of the marker. Moving the model to the desired position, an correction has to be added to the x and y values to center the model. Sometimes it’s also necessary to correct the z-coordinate, when a part of the model is projected behind the video stream.

**Transformation of the model**

Next the cone model has to be transformed according to the camera view and then fix this model on the marker. ARToolkit provides a transformation matrix which includes the position and orientation of the webcam and a struct containing marker information. Using Euler angles [Bourke00], it is possible to extract the rotations (pitch, jaw and roll) from this transformation matrix and use this to transform the object accordingly. From the struct the position of the detected marker can be extracted and this information to can be used to translate the model to the right position.

All the information needed to translate and rotate the model, can be found in the two ARToolkit variables: `patt_trans` and `marker_info`. To rotate the model around each of the x, y and z axis, the angles of the `patt_trans` matrix needs to be extracted to rotate the VTK actor.

The `patt_trans` matrix provided by ARToolkit is an 3x4 matrix. In the first three columns of this matrix, the rotation, sheers, translation and scaling values are all present. To extract the rotations, Euler angles are, which works as follows:
The last column of \texttt{patt\_trans} don’t provide any useful information now, so the remained transformation matrix is shown below:

\[
\begin{pmatrix}
  X_x & Y_x & Z_x \\
  X_y & Y_y & Z_y \\
  X_z & Y_z & Z_z
\end{pmatrix}
\]

To extract the three rotations the following formulas are needed:

Rotation around x: \( \text{asin} (-Y_z) \)
Rotation around y: \( \text{atan2} (X_z, Z_z) \)
Rotation around z: \( \text{atan2} (Y_x, Y_y) \)

Thus to calculate the angles, the elements of the transformation matrix can be used directly in these functions. For example, to calculate the rotation around x, the Yz element at (2,1) of the \texttt{patt\_trans} matrix needs to be passed to the \text{asin} function. The other two rotations can be obtained in the same way. At this moment the extracted angles are in radians, which needs to be converted to degrees before VTK is able to handle them correctly.

\[
\begin{align*}
\text{tx} &= \frac{\text{asin}(-\text{patt\_trans}[2][1]) \times 180}{\pi} \\
\text{ty} &= \frac{\text{atan2}(\text{patt\_trans}[2][0], \text{patt\_trans}[2][2]) \times 180}{\pi} \\
\text{tz} &= \frac{\text{atan2}(\text{patt\_trans}[0][1], \text{patt\_trans}[1][1]) \times 180}{\pi}
\end{align*}
\]

\section*{Scaling}

When using the approach above, the object is always the same size. Therefore a scaling function must be implemented. This way the object is bigger when the camera is closer to the marker. To implement this function, first the distance between the webcam and the marker must be calculated. Secondly an interpolation function must be written to obtain the best scaling at every distance.

To calculate the distance between the webcam and the center of the marker, first the difference between their x values (\texttt{diffX}) and y values (\texttt{diffY}) must be calculated. This must be done using \texttt{marker\_info} for the position of the marker’s center and elements of \texttt{patt\_trans} for the position of the webcam. The difference in z values is simply is the z position of the webcam: \texttt{patt\_trans[2][3]}, because the video stream, and thus the marker, is displayed at \( z = 0 \).

\[
\begin{align*}
\text{diffX} &= \text{abs}(\text{abs}(160-\text{marker\_info}\rightarrow\text{pos}[0]) - \text{patt\_trans}[0][3]) \\
\text{diffY} &= \text{abs}(\text{abs}(120-\text{marker\_info}\rightarrow\text{pos}[1]) - \text{patt\_trans}[1][3])
\end{align*}
\]

Now Pythagoras need to be applied to obtain the \texttt{groundLine}: the hypotenuse of x and z (see figure 8.1). Now this line is obtained, Pythagoras need to be applied once again to calculate the distance between the camera and the marker. This time, the hypotenuse of \texttt{groundLine} and \texttt{diffY} need to be calculated. This leads to the final variable \texttt{distance}:

\[
\begin{align*}
\text{groundLine} &= \sqrt{(\text{diffX} \times \text{diffX}) + (\text{patt\_trans}[2][3] \times \text{patt\_trans}[2][3])} \\
\text{distance} &= \sqrt{(\text{groundLine} \times \text{groundLine}) + (\text{diffY} \times \text{diffY})}
\end{align*}
\]
Now the two scaling values must be found in order to scale the model in the same proportions as the
distance between the webcam and the marker. To construct a right interpolation function, two extreme
distance values and their corresponding scaling must be calculated: one where the webcam is close
to the marker and one where the webcam is at a large distance from the marker.

This works as follows: First put the webcam close to marker, so that the marker fills almost the total
webcam picture. Of course, the application must detect the marker in that position. If not, then put the
webcam a little further away, so the marker is detected again. Let’s say, the obtained value of distance
is around 133 pixels. Then set the scaling of the object to the preferred size at such a close distance.
Let’s say that object is scaled 15 times. Memorize the size of the marker in relation to the model.

Now put the webcam at large distance of the marker; the distance has to be between the 600 and 1000
pixels away from the marker. Find the right scaling of the model, in such way that the model and the
marker have the same relative distance as in the first case. Let’s the found scaling is 2.5 at a distance
of 970 pixels.

At this moment there are two distances (let’s call them smallDistance and largeDistance for
now) and their associated scaling parameters (smallScale and largeScale). Between these two
distances an interpolating parabolic function for the scaling value must be calculated.

<table>
<thead>
<tr>
<th>distance</th>
<th>scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>133</td>
<td>15</td>
</tr>
<tr>
<td>970</td>
<td>2.5</td>
</tr>
</tbody>
</table>

Difference in distance (\( \text{diffDist} \)): \( 970 - 133 = 837 \)
Difference in scaling (\( \text{diffScale} \)): \( 15 - 2.5 = 7.5 \)

When distance is smaller than smallDistance, set the scaling to smallScale. The same goes for
when the distance is greater than largeDistance, then just set the scaling to largeScale.

After several trials, the result was that a linear interpolation function doesn’t lead to the right results.
When the distance is small, the scaling must increase faster then when the distance gets a higher
value. Instead of a linear interpolation function, a parabolic function must be used. To obtain the right
scaling values between smallDistance and largeDistance, the parabolic function below to must be used:

\[
\text{scaling} = \text{smallScaling} + \frac{(\text{diffScaling} / \text{diffDist}^2)}{(\text{distance} - \text{diffDist} - \text{smallDist})^2}
\]

The graph of this function is shown below:

![Graph of the parabolic function](image)

**Figure 8.2: Graph of the parabolic function**

**Full screen mode**

The application must display the grabbed webcam images full screen, irrespective of the resolution of the user’s monitor and the chosen webcam resolution. When the application starts, a dialog box appears in which the webcam resolution must be chosen.

![Property sheet of the webcam](image)

**Figure 8.3: Property sheet of the webcam**
When choosing a resolution, the size of the video stream in VTK will change accordingly. Therefore, to obtain a full screen video stream, the zooming function of the virtual camera must be changed for each different webcam resolution.

First the virtual camera must always focus to the center of the webcam images and the position of the camera itself must be positioned at a distance of 1000 pixels of the focal point:

```cpp
ren1->GetActiveCamera()->SetFocalPoint((double) xsize / 2, (double) ysize / 2, 0);
ren1->GetActiveCamera()->SetPosition((double) xsize / 2, (double) ysize / 2, 1000);
```

When using a webcam with resolution 320x240, the virtual camera must zoom with factor 2.2 to obtain a full screen image. When using a webcam with twice this resolution, the virtual camera must zoom with half this factor. Because this interpolation function is linear, you can calculate the right zooming value easily, when using the x size of the chosen webcam resolution, in combination with a predefined window size and the associated zooming value:

```cpp
ren1->GetActiveCamera()->Zoom(2.2 / ((double) xsize / 320));
```

### 8.1.3. Models

Four models were used to create the AR applications. The general principles of all these applications are discussed in the paragraph above. Because each model differ from each other, in this paragraph the details of each specific model will be explained in more detail: first the water flow around a boat’s hull, then the temperature inside an oven, next a model which representations energy flows inside a human’s brain will be discussed and at last a model of bubbles inside a human’s lung will be described in more detail.

**The boat**

This model represents the water flow around a ship’s hull. This model data is red line by line from a .DAT file and then the file is closed:

```cpp
if((input = fopen("barco.dat", "r")) == NULL)
    return;
while (fscanf(input, "%e %e %e %e %e %e %e\n", &x, &y, &z,&p,&vx,&vy,&vz) != EOF){
    tetraPoints->InsertPoint(n, x, y, z);
    vector->InsertTuple3(n,vx,vy,vz);
    escalar->InsertTuple1(n,p/1000000*256);
    n++;
}
if (fclose(input))
    return;
```

The part of the model that is of most interest is the flow around boat itself, but this is not the center of the model. Therefore the x and y coordinates of the model’s center must be corrected to set the boat itself on the middle of the marker, instead of the model’s center. Because the model was shown partly behind the video stream, the model also had to be moved upwards along the z-ax with 150 pixels:

![Figure 8.4: Complete model in full AR mode](image)
hedgeActor->SetPosition(marker_info->pos[0], marker_info->pos[1], 150);

When extracted the Euler angles from patt_trans, it’s clear that these rotate angles have a deviation concerning this model. To compensate this, the y-angle must be turned 180 degrees and the z-angle must be inverted.

hedgeActor->SetOrientation(tx, ty, -tz);
hedgeActor->RotateY(180);

The distance function for this model is taken between the 133 and 970 pixel and with scaling parameters between the 2.5 and 15.

// The scaling is calculated by measuring the distance closest to the // marker and far away and than make use of a parabolic function to // calculate the intermediary values.
if (distance <= 133)
    scaling = 15;
if ((distance > 133) && (distance < 970))
    scaling = 2.5 + (0.00001775767 * (pow(distance - 131-839, 2)));
if (distance >= 970)
    scaling = 2.5;

hedgeActor->SetScale(scaling);

This is the basic model that was at first used for testing usability. Eventually, four different versions of this model were constructed. These versions differed in the model itself and the background and the right version can be chosen by changing the defines in the head of the source code and then compile the application again.

When SCALED_VERSION is defined, only a subset of the full model which contains the part of interest of the boat will be constructed and this part is scaled, so the details will be much more present. This is done by taken only the coordinate model points (points have x, y and z coordinate values), which have an x coordinate value smaller than 25. This way the model is shortened only in its x direction and shows the full contents of the model in the y and z direction.

while (fscanf(input, "%e %e %e %e %e %e %e
", &x, &y, &z,&p,&vx,&vy,&vz) != EOF)
{
    // To make the model smaller we only draw the model points(x,y,z) from x = 1 until x = 24.
    #ifdef SCALED_VERSION
    {
      if(x < 25)
      {
        tetraPoints->InsertPoint(n, x, y, z);
        vector->InsertTuple3(n,vx,vy,vz);
        escalar->InsertTuple1(n,p/1000000*256);
      }
    }
    #else
    {
        tetraPoints->InsertPoint(n, x, y, z);
        vector->InsertTuple3(n,vx,vy,vz);
        escalar->InsertTuple1(n,p/1000000*256);
    }
    #endif
    n++;
}
And to change the scaling of the object, the code below is used:

```c
#ifdef SCALED_VERSION
{  
  if (distance <= 133) scaling = 15;
  if ((distance > 133) && (distance < 970))
    scaling = 2.5 + (0.00001775767 * (pow(distance - 131-839, 2)));
  if (distance >= 970) scaling = 2.5;
}
#else
{  
  if (distance <= 133) scaling = 9.5;
  if ((distance > 133) && (distance < 815))
    scaling = 1.5 + (0.0000202848 * (pow(distance - 133-628), 2)));
  if (distance >= 815) scaling = 1.5;
}
#endif

To view the model from the side, only a rotation of 90 degrees is needed to accomplish that:

```c
#ifdef TURN_90_DEGREES
hedgeActor->RotateX(-90);
#endif
```

Then there are a couple of different settings for the video stream, which also can be changed by defining one of the following options in the head of the source code. For example, a representation without any marker and camera image, having a totally black background, can be obtained by defining TOTALLY_BLACK.

```c
#ifdef VIDEO_STREAM
imageActor->SetVisibility(1);
#endif

#ifdef TOTALLY_BLACK
imageActor->SetVisibility(0);
#endif
```

During the experiments with users, it became clear that the user could get lost, because there were no visible markers to guide the user. After experiments described above, a solution for this problem was found: to implement a thin white line around the marker. The struct `marker_info->vertex` contains the x and y coordinates of the four corners of the marker. The upper left corner coordinates are stored in element 0,0 and 0,1 the upper right corner coordinates in element 1,0 and 0,0. The lower left coordinates are in 2,0 and 2,1 and the lower right in 3,0 and 3,1. Those four point are connected to each other by four white lines. When ARToolkit detects the marker these four lines are drawn. Below only the topline is shown:

```c
// Show the borders of the marker
topLine->SetPoint1(marker_info->vertex[0][0], marker_info->vertex[0][1], 0);
topLine->SetPoint2(marker_info->vertex[1][0], marker_info->vertex[1][1], 0);
```

![Figure 8.5: Marker represented as white line](image-url)
By default, the white lines are made invisible, but if `WHITE_LINES` is defined, this code will be executed:

```c
#ifdef WHITE_LINES
    imageActor->SetVisibility(0);
    topLineActor->SetVisibility(1);
    leftLineActor->SetVisibility(1);
    bottomLineActor->SetVisibility(1);
    rightLineActor->SetVisibility(1);
#endif
```

Additionally, a new version was created in which the video stream is only visible when the user loses the marker, so it can be found again more easily:

```c
#ifdef VIDEO_WHEN_LOSS
    imageActor->SetVisibility(!markerFound);
#endif
```

Finally we also added stereoscopic vision, which was added in the `vtkInit` function:

```c
renWin->StereoCapableWindowOn();
renWin->SetStereoTypeToCrystalEyes();
renWin->StereoRenderOn();
```

**The oven**
The next model was a model of the temperature in an oven. The biggest difference with the boat model was that the coordinates of this model were stored in another format than the previous model. To visualize the data we wrote some additional code to construct a proper VTK actor. Because this dataset wasn’t used in the experiments, it won’t add any significant information to discuss it in detail here.

![Figure 8.6: Representation of temperature inside an oven](image)

**The brain**
The third model was a model which represents energy flows inside a human’s brain, using vectorial dipole data. This application that belonged with this model allowed keyboard interaction to interact with the model. These required additional keyboard interaction required VTK 5.0, thus switching to this newer version of VTK was necessary. From this model where also made four different versions resembling the boat model. The first had the normal video stream on the background, the second had a totally black background, and the third a black background with four white lines and the last show only the video stream whenever the marker is undetected. The adjustments concerning the last three
models are similar to those of the boat model. A detail explanation of this can be found in the paragraph of the boat model.

This model exists of two different models. The first is called Dipolos and shows the arrows representing the energy flow in the brain. The second one is Headmodel, which shows a transparent overlay on top of the Dipolos model, which is actually a hull of the brain. Thus the transformation needs to be applied to both models.

The centre of the brain model isn’t on the middle of the marker and the Euler angles have a slight deviation. This needs to be corrected on both models:

```cpp
dipolos->getActor()->SetPosition(marker_info->pos[0] - 5, marker_info->pos[1] - 7, 100);
dipolos->getActor()->SetOrientation(tx, ty, -tz);
dipolos->getActor()->RotateX(180);

headmodel2->getTransparentHeadModelActor(0.8)->SetPosition( marker_info->pos[0] - 5, marker_info->pos[1] - 7, 100);
headmodel2->getTransparentHeadModelActor(0.8)->SetOrientation(tx, ty, -tz);
headmodel2->getTransparentHeadModelActor(0.8)->RotateX(180);
```

The distance function of this model is also similar to the boat. The distance values are 140 and 950 pixel with scaling parameters between 0.2 and 1.5:

```cpp
if (distance <= 140) scaling = 1.5;
if ((distance > 140) && (distance < 950)) scaling = 0.2 + (0.0000019814052735863435451912818 * (pow(distance - 810 - 140, 2)));
if (distance >= 950) scaling = 0.2;
```

This model includes keyboard interaction. It is possible to switch to other energy flow patterns and to remove the legenda because this can be found unpleasant when looking in stereoscopic vision. This kind of interaction is handled by callbacks:

```cpp
virtual void Execute(vtkObject *caller, unsigned long, void*)
{
    char key = iren->GetKeyCode();
    switch (key)
    {
    case '+':
        dipolos->constructDipolos(dipolos->timelineNext());
        renWin->Render();
        break;
    case '-':
        dipolos->constructDipolos(dipolos->timelineBack());
        renWin->Render();
        break;
    case 'm':
    case 'M':
        // Code magnetude change
    case 'l':
    case 'L':
        // Code hide en show legenda
    }
}
The lungs

The last model represents bubbles inside a human lung. This model doesn’t have any interaction. There are three different versions to view the model. These versions only differ in the background opacity. The first version has a normal video stream at the background. This is equal to an opacity of null percent. The second had an opacity of twenty percent and the last one of hundred percent. Changing between these different versions can be managed by setting the right define:

```c
#ifdef VIDEO_STREAM
    imageActor->SetVisibility(1);
#endif

#ifdef TOTALLY_BLACK
    imageActor->SetVisibility(0);
#endif

#ifdef LOW_VIDEO_OPACITY
    imageActor->SetOpacity(0.2);
#endif
```

The origin of the long model was at the bottom, which means that the model rotates around this point at the bottom. Correcting this is needed:

```c
lungVolume->SetPosition(0, 0, 0);
lungVolume->SetOrigin(115, 125, 0);
```

This model also isn’t placed on the middle of the marker and the Euler angles also have a slight deviation. To correct this, the following code is needed:

```c
lungVolume->SetOrientation(tx, ty+180, -tz);
lungVolume->SetPosition(marker_info->pos[0]-120, marker_info->pos[1]-120, 0);
```
The distance function of this model is also similar to the boat. The distance values are 150 and 560 pixel with scaling parameters between the 0.6 and 2. A more detailed explanation of the implementation of this function can be found in the paragraph describing the boat data in this chapter.

8.2. Implementation of the VR application

In this part the implementation of the VR application will be explained. Here the chronological order of implementation will be followed. To have a clearer understanding of the available functions in each method, a complete UML diagram is displayed in Appendix A.

8.2.1 Loading a BSP level

For indoor levels, a commonly used format in various computer games is the BSP format. Loading a BSP level into OGRE is possible through the BspSceneManager, a SceneManager that is capable of reading levels with the BSP format. For testing purposes, the first BSP map that has been made for this application, was just a simple room in which the user should eventually be able to walk. All code for loading is done within the MazeApplication class, in the method setup, which calls SetupResources, chooseSceneManager and loadResources.

First, in SetupResources all locations for the resources (textures, models, scripts) will be loaded from the configuration files resources.cfg for Ogre-specific files and quake3settings.cfg for level-specific files. All resources for the level are stored in one file, maze.pk3. This is a zip pack containing all files needed for the level, such as the BSP file and all textures.

After the setup has been done, in chooseSceneManager the SceneManager will be created. By giving the OGRE Root object the name of the SceneManager we want, we get a SceneManager returned that has the capabilities of the BSPSceneManager.

Then, in loadResources, the world geometry will be loaded from the BSP map and all resources will be loaded into memory. After this, the loading has been completed and OGRE is able to display the level defined in the BSP map.

The application now already allows for moving within the level. This is possible because the standard structure of an OGRE example application has some code to move the camera. This of course is very basic and does not allow a user to play a game, since there is no game logic or collision detection yet.

8.2.2 Collision detection

OGRE itself is just a graphical rendering engine, not a game platform. So, additional things such as collision detection have to be added afterwards. This has been demonstrated in the OGRE Reference application, which uses ODE, a physics engine that not only detects collisions, but also can calculate bouncing objects or objects pushing each other. These features are not needed necessarily but might always prove themselves useful in a later version. Because the Reference application already demonstrates how to use ODE, this will be a great example to follow in this project. The iterative development process allows for changes in the design and the way details are worked out, which is needed for quick adaptation to previously unknown implementation details of parts of the source code of OGRE or the Reference application.

The first that has been done, is improving the way the Reference application is implemented, so the final game can be compiled as an individual project with clearly separated objects. This was a time-consuming but necessary step, since the original code of the Reference application was not separated in .h and .cpp files, and the construction of new objects was not as clear as it could be. With the revised code, the application has a more consistent look and is easier to maintain. For example, a separate Obj_Player is created to define a physics object for the player that should be able to detect collisions with the level.

The collision detection in the Reference application however did not detect collisions of the camera or player object against the level geometry. The first idea was that this problem was caused by the ODE
physics engine. After testing a new implementation with the Opcode collision detection, this seemed not to be the problem. It was caused by to some restrictions the BSPSceneManager has in its implementation. Because BSP is a format that is a little outdated and not used a lot any more, the OGRE implementation only allows for the most basic BSP support. More advanced functions need to be implemented separately, using the structure of the Reference application as a guideline.

After the cause of collision detection problems had been found, ODE will be used again, since object-to-object collision detection and physics simulation may be useful in later stages of the game. Due to the limited time before delivering a first version of the game, the object-to-level collision detection is now implemented as a “functional proof of concept”. Here the implementation of this proof of concept will be explained.

Floor collision detection
The collision detection at this moment consists of two separate parts: the first part takes care of keeping the player down to the floor by some gravity, the second part takes care of collisions of the player with the walls in the level. Both are based on the same functions of the SceneManager.

First, to detect a floor, the application has to “look down” and see what objects are beneath the player and make a list of them. This is done with the following code inside the MazeFrameListener, called each frame inside the FrameStarted method:

```cpp
static Ray updateRay;
updateRay.setOrigin(mPlayer->getPosition());
// look down
updateRay.setDirection(Vector3::NEGATIVE_UNIT_Z);
mRaySceneQuery->setRay(updateRay);
// try sorting results, however this doesn't always work
mRaySceneQuery->setSortByDistance(true, 0);
RaySceneQueryResult& qryResult = mRaySceneQuery->execute();
```

Here the application starts with looking down the Z-axis (which represents the height axis in a BSP level). All objects that are found, will be added to the qryResult variable. See also figure 8.9 below, which displays the player object 100 units above the first object found. This distance is needed, because the camera is directly attached to the player, so the player location is the point where he is looking from, and has to represent the height of the players eyes.

![Figure 8.9: example of a RaySceneQuery with the floor and objects beneath the floor](image)

The command “setSortByDistance(true,0)” should make the query automatically sort all results, but OGRE’s implementation of the RaySceneQuery with BSP objects doesn’t do this always. So, to get the lowest distance measured, the entire RaySceneQueryResult has to be searched. This is done with the following code:
RaySceneQueryResult::iterator i = qryResult.begin();
Real toMove = 0.0;
// loop through result, because automatic sorting isn't always correct
while (i != qryResult.end()){
    // avoid weird, sometimes occurring bug in ogre/BSPSceneManager
    // by just ignoring values that are too low
    if (toMove == 0)
        if (i->distance > PLAYER_HEIGHT*0.7) toMove = i->distance;
    else
        if (i->distance > PLAYER_HEIGHT*0.7)
            toMove = (toMove < i->distance) ? toMove : i->distance;
    i++;
}
if (toMove == 0) toMove = mLastHeight;
else mLastHeight = toMove;

The variable toMove will contain the distance the player has to move to touch the first object, but when just taking the lowest value from the qryResult, this occasionally returns values that are way too low, probably caused by some bugs or implementation flaws in the BSPSceneManager. This problem can be solved by just ignoring all values that are less than 70% of the players height and just returning the height OGRE has returned in a previous frame. This is a robust fix, since it is very unlikely a player will have to move up by more than 30% of his own height in just one frame. Only in very low framerate situations, this might occasionally lead to problems of the player falling through the floor.

After the distance of the player to the floor has been determined, the application has to calculate the distance the player will have to move upwards or downwards. This code first fixes an unsupported function within the BSPSceneManager. Let’s first shortly explain this problem, before moving on with the code.

![Figure 8.10: lookout point, covered with a full clip](image)

In the final level, there will be a ‘lookout point’ (shown in figure 8.10), from which the player can look down into a hallway. This place is covered with a “full clip” object, which normally should not be rendered but still detected with collision detection, not allowing the player to pass through. However, the OGRE implementation of the BSP format just ignores the “full clip”, resulting in this object not being rendered, but also not being detected by the RaySceneQuery. This fix, actually fixing a flaw in the code for the wall detection, has to take place in the code for the floor, because it couldn’t be detected by the code for the walls. It consists of only one line of code moving the player back to his old position, and is selfexplanatory.
// avoid falling from too high objects (level specific bugfix, due to BSP clipping object not supported by the Ogre BSPSceneManager.)
if (toMove > 330) {
    mPlayer->setPosition(mOldPos);
} else {  // place user at new position above floor

When the player isn't at this specific place of the level, the code moves on with placing the user at the desired height above the floor. Again, this code isn't how it should be done in a final version, but serves quite well as a proof of concept.

    if (toMove > PLAYER_HEIGHT*1.5 && !(mPlayer->falling()) && !(mPlayer->jumping())) {
        mPlayer->logEvent(mTimer->getMilliseconds(), "fall", toMove);
        mPlayer->doFall(toMove-PLAYER_HEIGHT);
    }
    if (mPlayer->jumping() && toMove > PLAYER_HEIGHT) {
        toMove = toMove - mPlayer->getJumpHeight();
    }
    if (mPlayer->falling() && toMove < PLAYER_HEIGHT && toMove > 15.0) {
        mPlayer->doFall(0);
    }
    mPlayer->setPosition(  
        mPlayer->getPosition().x,  
        mPlayer->getPosition().y,  
        mPlayer->getPosition().z - toMove + PLAYER_HEIGHT + mPlayer->getJumpHeight() + mPlayer->getFallHeight());

The first part of this code will check if the player has to be placed directly at the new position (simulating a player walking down a slope), or if the player has to be moved in a “smooth” movement, simulating a player falling down from a certain height. Next, it will be checked if the moving distance has to be changed if the user is jumping, and the fall height of the player will be reset if the player is going to fall lower than his own height (all fall- and jump heights are being calculated by the player-object, which will be discussed later in this chapter). The last line will put the player at his new height above the floor, taking his own height, the jump height, and the fall height into account.

Wall collision detection
The detection of the player colliding with the wall is done almost the same way as the keeping the player to the floor, but now we have to make use of the direction the player is moving in. This will also take place in the MazeFrameListener, but this time in the MovePlayer method, because the direction of movement is only available after the user input has been processed. This is how the RaySceneQuery is made for the player to determine if some objects are on his way when he is moving:

    mOldPos = mPlayer->getPosition();
    mPlayer->setCurrentVelocity(currentVelocity);
    mPlayer->translate(currentVelocity*mMoveScale);
    // update statistics within player object
    mPlayer->countDist(currentVelocity.length()*mMoveScale);

    static Ray updateRay;
    updateRay.setOrigin(mPlayer->getPosition());
    updateRay.setDirection(mPlayer->getPosition() - mOldPos);
    mRaySceneQuery->setSortByDistance(true, 0); // doesn't always work!
    mRaySceneQuery->setRay(updateRay);
    RaySceneQueryResult result = mRaySceneQuery->execute();
After the QueryResult is acquired, the lowest value might not be in the first element, for the same reason as with the floor detection: the code of OGRE sometimes returns an incorrect value. So getting the lowest distance is done by the next simple loop:

```cpp
while (ri != result.end()){
    if (ri->worldFragment!=NULL){
        if (lowest == 0)
            lowest = ri->distance;
        else
            lowest = (lowest < ri->distance) ? lowest : ri->distance;
    }
    ri++;
}
```

The code for this collision detection is, like the floor detection, also a proof of concept. This very simple implementation just puts the player back to his old position whenever he is getting too close to an object:

```cpp
if (lowest!=0 && lowest <= COLLISION_DISTANCE){
    mPlayer->setPosition(mOldPos);
    // log event if this is a new collision
    if (mTimer->getMilliseconds()-mLastCollTime>200)
        mPlayer->logEvent(mTimer->getMilliseconds(),"collision");
    mLastCollTime = mTimer->getMilliseconds();
    currentVelocity = Vector3::ZERO;
    mPlayer->setCurrentVelocity(currentVelocity);
    lowest = 0;
}
```

This simply disallows the player to move into the direction of the wall, but does not provide a “natural” feedback as one might be used to in most games. Normally the player should slide along the wall, in the direction he is pointing. This simple implementation doesn’t support this, but serves well as a proof of concept for collision detection within a BSP level and is working stable enough for a first version of the gaming environment.

### 8.2.3 The Player object

As already shown in the explanation of the collision detection before, the player object has some internal code to calculate falling, jumping and logging the statistic data which will be needed by the experiments at IEETA. These methods will be explained here.

First, the starting method of the player object is the `setup`-method, which is automatically called by the constructor. In this method the player will be assigned a physical object and all dynamics properties will be set (if eventually needed, but not used for this version).

When the player has to fall or jump, the MazeFrameListener calls `doFall`, this will just set the `mFallHeight` variable. When the `doJump` method is called, the `mMaxJumpHeight` and `mJumpCorrection` values will be set in this way:

```cpp
mMaxJumpHeight = height;
mJumpCorrection = -height;
```
The real calculation is done within the `renderHeight` method, which the MazeFrameListener will call each frame. This is how the new heights are rendered in this method:

```c++
// calculate new fall height
mFallHeight = mFallHeight-(PLAYER_FALLSPEED*timeSinceLastFrame);
if (mFallHeight < 0)
    mFallHeight = 0;

// calculate new jump correction
mJumpCorrection = mJumpCorrection
    + (PLAYER_JUMPSPEED*timeSinceLastFrame);
if (mJumpCorrection>mMaxJumpHeight){
    mJumpCorrection = 0;
    mMaxJumpHeight = 0;
}
```

And the method to get the new jumping height has just one line:

```c++
return mMaxJumpHeight - (std::abs(mJumpCorrection));
```

As can be seen from the methods above, the calculation of these variables will be linear, not yet representing a realistic simulation of a falling or jumping object. However, since this code will only be used very occasionally, this is not an issue that would need improvement at this first version of the game. The code above is quite selfexplanatory: since this is just a simple approximation of real physics, the player will just make a linear \ movement when falling, or a \ movement when jumping.

The player does not only calculate its falling and jumping heights, but also keeps track of all statistical data that has to be saved after the user finishes playing the game. The total distance travelled will just be counted in an integer value, but other player events (falling, finding a target, or colliding with a wall) will be saved in a vector of PlayerEvent structures. The PlayerEvent structure is saving three variables and is defined in `Obj_Player.h`:

```c++
struct PlayerEvent{
    long mEvtTime;
    char* mEvtName;
    Real mEvtValue;

    PlayerEvent() {};
    PlayerEvent(int _t, char* _n, Real _v) {
        mEvtTime = _t;
        mEvtName = _n;
        mEvtValue = _v;
    }
};
```

The player will save every event entered by `logEvent` to an array of elements of this structure:

```c++
PlayerEvent pE = PlayerEvent(evtTime,evtName,evtValue);
mEvtList.push_back(pE);
if (!strcmp(evtName,"target")) mTargets++;
```

This makes the types of events that can be logged very flexible, every different name can be attached to `evtName`, allowing a next version of the game to log even more events if needed.
Saving the event log simply takes place in the MazeFrameListener, which calls the `saveEvtLog` method of the player object. This method will save everything in a logfile, which can be used for the research at IEETA. An example of a logfile can be found in Appendix C.

Finally, a little change to some code had to be made to make the movement of the player stable: since a mouse only has two axes (x and y), only two rotations could be made (roll and pitch). However, the default movement would be like an airplane, which also yaws when making a turn. Making the user always look straight requires the player object to have a fixed yaw axis, an option that was not supported by the ApplicationObject class. Some lines of code have been added to the setOrientation function to add this functionality:

```cpp
Quat q = Quat(orientation);
if (mFixedYawAxisEnabled)
    Radian y, p, r;
    Mat3 m;
    orientation.ToRotationMatrix( m );
    m.ToEulerAnglesZYX( y, p, r );
    // only tested UNIT_Y, others may need other Euler Angle ordering, but did not test this!
    if (mFixedYawAxis==Vector3::UNIT_X)
        m.FromEulerAnglesZYX( Radian(0.0f), p, r );
    if (mFixedYawAxis==Vector3::UNIT_Y)
        m.FromEulerAnglesZYX( y, Radian(0.0f), r );
    if (mFixedYawAxis==Vector3::UNIT_Z)
        m.FromEulerAnglesZYX( y, p, Radian(0.0f) );
    q = Quat( m );
}
mSceneNode->setOrientation(q);
```

This code sets one of the rotation axes to zero. A simple but effective solution to keep the player movement steady. When using the tracker, this function will not be needed, since the tracker provides the software with three rotation axes. Therefore, the ApplicationObject also has a new function called `setFixedYawAxis`, which can simply enable or disable this fixed axis.

### 8.2.4 The targets

To begin with, the locations of the targets first have to be saved somewhere. The easiest way to do this, is simply adding them as an entity inside the BSP file. This can simply be done in the editor GTKRadiant. However, it was impossible to read the locations of those entities out of the file, since the OGRE BSPSceneManager was only supporting very basic functions of the BSP file. Reading entities was not one of them, so this function had to be added to the BSPSceneManager. Since OGRE is distributed as a collection of dll and linkable library files, the OGRE source had to be downloaded and edited. This was not a problem, since OGRE is Open Source. The addition of this function involved editing of multiple files. A complete overview of the files which had to be edited within the OGRE source can be reviewed in Appendix D.

Now, the targets can be loaded, which will be done in the `SetupScene` method of the MazeApplication class. Since the SceneManager has been edited and a simple `CreateBox` method is available within the ObjectManager, this can be done quite easily with the next code.

![Figure 8.11: A target inside the game](image)
// now load entities
int numEnt = mSceneMgr->getEntityCount();

for (int i=0;i<numEnt;i++){
    if (mSceneMgr->getEntityData(i).entName=="target"){  
        Maze::Box* mBox = mObjectManager->
            createBox("target"+StringConverter::toString(i),
            15,15,15,mPlayer,mTimer,
            mSceneMgr->getEntityData(i).vp.position);
        mBox->setCollisionEnabled(true);
        mBox->getEntity()->
            setMaterialName("Examples/10PointBlock");
    }
}

The used texture in the code above, Examples/10PointBlock, is just a default texture which is provided by the OGRE SDK. You can see how a target would look like in figure 8.11.

As you can see, this code is not too complicated. The only thing that might need some explanation, is where the attributes to the CreateBox method. First is the name and then three dimensions. Next, the Box object will need to know the pointers to the Obj_Player and Timer objects, so these need to be added to. Finally, the location of the target is given.

So what will the Obj_Box do with these player and timer objects? The box object simply consists of only a few methods. The first method, setup, will do the same as the setup method in the Obj_Player does: assigning a physical object and setting all dynamics settings for future use. The only other method in the Obj_Box class is _notifyCollided. This method is the only method in the entire game code which will be used by the ODE dynamics engine when a collision is detected, since this is the only object-to-object collision possible in the game right now. The box object does not need to do that much though, when it has collided with a player:

    mPlayer->logEvent(mTimer->getMilliseconds(),"target");
    this->getEntity()->setVisible(false);
    this->setCollisionEnabled(false);
    char soundfile[] = "..\..\media\sounds\tada.wav";
    PlaySound(soundfile,NULL,SND_FILENAME|SND_LOOP|SND_ASYNC);  

This is all: let the player log the time and the event which happened, hide this target box and don’t listen to any collisions anymore, and finish with playing a simple sound. Since there was no time to implement a sound engine into OGRE, this short line using a builtin function of windows.h will do the job for now. In a later stage, the Box object might play its sound through a sound engine object.

8.2.5 Controlling the game

All functions that will be used to control user movement within the game, is contained in the MazeFrameListener class, since this class has methods which will be called each time a frame has to be rendered. First the individual data acquisition will be explained for each input device, then the final calculation of movements will be explained.

Keyboard control

The user will be able to walk forward and backward using respectively the W and S keys, as used in most computer games. Also strafing to the left and right will be possible, using the A and D keys. However, it might be desired by the experiment not to strafe when pressing the A and D keys, but to turn the user around. This behaviour can be configured by the user. The input of the keys will be captured in the processKeyInput method, adding the movements to the variable called
mTranslateVector, which will eventually contain the sum of keys pressed that frame. For example, the code for one key will look like this:

```cpp
if (mInputDevice->isKeyDown(KC_W)) {
    // walk FORWARD
    mTranslateVector.z = -mMoveSpeed;
}
```

One special key is also added, but eventually not needed for playing the game:

```cpp
// jumping .. needed or not?
if (mInputDevice->isKeyDown(KC_SPACE)) {
    if (!mPlayer->falling() && !mPlayer->jumping()) {
        mPlayer->doJump(30.0);
    }
}
```

### Mouse control

The movement of the mouse will be used the same way as this is done in most computer games: mimicking the movement of the players head. First, the mouse input will be captured in `processMouseInput`, saving the movements in `mRotX` and `mRotY`:

```cpp
mRotX = MOUSE_SENSITIVITY
* Degree(-mInputDevice->getMouseRelativeX() * 0.13);
mRotY = MOUSE_SENSITIVITY
* Degree(-mInputDevice->getMouseRelativeY() * 0.13);
```

### Tracker control

The tracker will exactly give the orientation of the users head by giving the roll, pitch and yaw of the tracker. These variables should be saved each time into variables like “mLastRoll”, because OGRE should not keep turning the player when he is standing still. Only the difference of the new roll/pitch/yaw between the last ones will be added to the movements of the player in the game. Aquiring these angles is done in the `FrameEnded` method, just before the `MovePlayer` method will be called:

```cpp
if (mSettingUsingTracker) {
    if (IS_handle>0) {
        ISD_GetData( IS_handle, &IS_data );
mYaw = IS_data.Station[0].Orientation[0];
mPitch = IS_data.Station[0].Orientation[1];
mRoll = IS_data.Station[0].Orientation[2];
    }
}
```

The tracker is shipped with a well documented SDK, so aquring this data is done with very little code and does not need creating a new object. The tracker is created in the constructor of the MazeFrameListener, using some lines of code, but this one is the most important:

```cpp
IS_handle = ISD_OpenTracker(NULL, 0, FALSE, FALSE );
```

This is all it takes to enable the tracker, switch it on, and get a handle returned.

### Accelerometer control

Implementing support for the accelerometer was a little harder than implementing support for the tracker. The accelerometer was manufactured as a project of the University of Aveiro, and is not supported with a well documented SDK. However, this accelerometer had already been used in an
other project at IEETA, called the RVirtual application. Here, the accelerometer is used as a “joystick”, the same way it should be used in this game. The code used to read out the serial port and have access to the accelerometer could be partly reused in this application.

When first trying to add the existing methods into the Maze game, it worked out quite useless: the maze game made use of windows.h, and the accelerometer used a multithreading structure which forbids the usage of windows.h. This made the multithreading code totally incompatible with the structure of the game. After trying the sensor without multithreading, it proved useless: windows locked up during usage of the serial port, making the frame-per-second count of the game drop dramatically. After some research, an other form of multithreading has been found and implemented to the sensor code, making it support multithreading in such a way that it was compatible with the code of the game. Showing only one part of the code here should roughly explain this principle.

```c++
while(pollStop==false) {
    // lock
    WaitForSingleObject(hRunMutex, INFINITE);
    // Read sensor data
    serial->Flush();
    serial->Write("a",2);
    serial->Read(&buffer, size);
    // Parse data
    string parsebuff(buffer);
    ParseString(parsebuff, xyz, dxyz);
    ReleaseMutex(hRunMutex);
    // give other functions a moment to reach the variables
    Sleep(15);
}
```

The code above is the loop in which the serial port will be read. The two most important lines are those with the `hRunMutex` variable in it. The first one will lock the object so that other threads could not reach it, the last one releases the object, giving other threads a little time to read the variables inside the SerialSensor object. This could be done by calling the `getTranslateVector` method for example, which simply returns a translation vector in the same format as the keyboard code would do.

After several tests, it became clear the accelerometer wasn’t an input device that allowed smooth browsing within the levels. The input was sometimes a little delayed, and the accelerometer sometimes seemed not to respond quite well. Controlling the game with a keyboard while having the HMD on wasn’t comfortable too, since the user could stand backwards in front of the keyboard. This problem has been solved by using an “alternative accelerometer”: keeping the mouse in a hand and using the two buttons for controlling walking forward and backward. This is implemented by just adding these lines of code, which just add to the translate vector like a keyboard input would normally do:

```c++
if(mSettingUsingTracker && mInputDevice->getMouseButton( 0 ))
    mTranslateVector.z = -mMoveSpeed;
if(mSettingUsingTracker && mInputDevice->getMouseButton( 1 ))
    mTranslateVector.z = mMoveSpeed;
```

**Calculation of movements**

When the input from the chosen devices has been processed, the player has to be rotated and moved into the desired direction. This is done in the `movePlayer` method, done in two steps. First the rotation will be executed, then the walking movements will be executed (and the earlier explained collision detection code will be applied after this).

For each mode of control, the application needs to behave differently. For example when the user is using the tracker device, the application has to use the yaw, pitch and roll variables:
if (mSettingUsingTracker){
    //using tracker input
    if (mSettingTurnStrafe){
        mCamera->yaw(-(mYaw-mLastYaw));
        if (mSettingUsingAccelerometer){
            mPlayer->yaw(Degree(-0.15 * mTranslateVector.x));
        } else{
            mPlayer->yaw(Degree(-1.8 * mTranslateVector.x));
        }
    } else{
        mPlayer->yaw(-(mYaw-mLastYaw));
    }
    mCamera->pitch(mPitch-mLastPitch);
    mCamera->roll(-(mRoll-mLastRoll));
    mLastPitch = mPitch;
    mLastYaw = mYaw;
    mLastRoll = mRoll;
} else{
    //using the mouse input

Since the code for the mouse input looks the same, only using the mRotX and mRotY instead, only explaining the code above should make the principle clear. The code has only one difference, since the mouse only has two rotation directions and a human head can freely move in all three directions, the code for the mouse will not include a line for a roll operation.

So, what do we see in the code above? The pitch and roll are the same for every option, but when the user has to turn instead of strafing, the yaw will be processed differently. When the user wants to turn by pressing the keyboard, the user has to be able to look around freely while moving in an other direction. This has been done by yawing only the camera object with the tracker input, and yawing the player object with the mTranslateVector input. If the user uses the accelerometer, it turned out he would need an other conversion factor than a player using the keyboard for input. After this, the translate vector in strafing direction will be set to zero. If the user doesn’t want to turn, but wants to strafe, the translate vector will not be reset, and the entire player object will be turned into the direction the user is looking at. This results in the player walking always in the direction he is looking at.

8.2.6 Configuration

To enable the researchers at IEETA to configure the game to their needs, they would like to have an option to save the logfiles to different names and have an option to switch the use of the tracker, accelerometer, and turn-instead-of-strafing on or off. OGRE has a builtin support for configuration files, so this has been implemented in a few lines:

```cpp
ConfigFile cf;
cf.load("maze.cfg");
mSettingFileName = cf.getSetting("FileName");
if (cf.getSetting("UsingTracker")=="yes"
 || cf.getSetting("UsingTracker")=="Yes")
    mSettingUsingTracker = true;
else
    mSettingUsingTracker = false;
```
For all options the same if-else construction as used for \textit{UsingTracker} has been used. The configuration file just contain options saved like this:

\begin{verbatim}
# Number of the logfile to save, don't forget to change each time!
FileName=demo

# Most needed switch: do you want the HMD Tracking devide to be used,
# or are we using the mouse for movements of the head?
UsingTracker=No

# Do you want the Accelerometer to be used? If not, when the Tracker
# is used, the mousebuttons can be used to walk forward and backward.
# !!!Warning: the application will not run stable when switched to
# Yes!!!
UsingAccelerometer=No

# Pressing A/D or tilting the accelerometer normally makes the player
# strafe in the game. If you want this to make the player turn
# instead, switch this setting to Yes.
TurnInsteadStrafing=No
\end{verbatim}

This allows a researcher to enter all configuration details before running the game, since this is the easiest way to implement all this without an extensive graphical user interface in the game.

\subsection*{8.2.7 Multiple program states}

During the implementation of the game, a little start has been made by trying to couple CEGUI with the application. CEGUI would be a graphical user interface with a lot of functionality. Unfortunately there was not enough time to make this an operational user interface. At one of the last days of implementation, there was some time to make a very simple user interface, using the OGRE overlay manager. An overlay doesn’t have many options, but this was enough to display the time left and the number of targets found. Also some messages could be displayed, saying the user found all objects, or ran out of time and is gameover.

When displaying a message, the application needs to stop playing the game and wait at other user input. This is done by the introduction of different program states in the application. For example, when the application starts, it has to check if the defined name for the logfile already exists. This is the abbreviated code for this:

\begin{verbatim}
if(fopen (filename, "r") == NULL)
  mProgState = PS_PLAYING;
else{
  mMessageOverlay->show();
  // some error message code here
  mProgState = PS_FILEEXISTS;
}
\end{verbatim}
When the frameEnded method wants to call the functions for processing the input and moving the player, it first checks if the \textit{mProgState} is equal to \texttt{PS\_PLAYING}. When it is not, the function \texttt{processKeyDialog} will be called, a function that handles the interaction of the user with dialog screen. In this example, the error message would ask if the user wants to continue overwriting the file, and \texttt{processKeyDialog} waits for an answer to this question. If the user wants to continue, this code would be executed, finally setting the program state to \texttt{PS\_PLAYING}.

```c
if( mInputDevice->isKeyDown( KC_C)) {
    // [C]ontinue is pressed, so reset the counter and start the game.
    mMessageOverlay->hide();
    mTimer->reset();
    mProgState=PS\_PLAYING;
    return true;
}
```

![Figure 8.12: The dialog screen appearing if a logfile already exists](image)

Figure 8.12: The dialog screen appearing if a logfile already exists
9. Experiments

At this moment, all experiments with the AR application at IEETA have been finished and most of the experiments with the VR application have been finished. Both applications have proven themselves useful during these experiments, but unfortunately we can not show the results yet. Interpreting all test results will start Monday the 26th at IEETA, and we hope to get some conclusions as soon as possible, so we can hopefully process and send them to our supervisors.
10. Conclusions

This chapter covers the conclusions of both applications. First the conclusions of the AR project will be described and then the conclusions of the VR project.

10.1. Conclusions for AR application

The visualization of complex VTK models in combination with the AR capabilities of ARToolkit produces a much more detailed and clear visualization of data than ARToolkit with VRML-models, which were used so far.

Stereoscopic vision is made possible by a VTK library. This gives the model a more realistic sense of depth of the model.

Although the visualization part of VTK is significant better, there are some drawbacks, which occur at each model.

During the implementation phase it became clear that the structure of each model differs a lot from each another. For example, some model data coordinates are given in a .DAT format while others are in .IMG format. This kind of differences makes it almost impossible to load models into VTK, just by entering their file name.

Each model has a different size, so every time the scaling and associated parabolic function needs to be recalculated manually for each model. In Appendix B, this is explained in detail.

Although these drawbacks for loading each VTK model correctly (which shouldn’t be so hard, when following the manual) this configuration still is a significant improvement over the previous used methods.

10.2. Conclusions for VR application

Implementing a game using the OGRE framework really helps developers saving some time. However since this is not a gaming platform, adding additional functionality might take some time if it does not work right from the beginning.

The OGRE implementation of the BSP SceneManager has some flaws and omissions, making it quite hard to get collision detection and entity readout working correctly.

Developing a game consists of many different tasks, requiring different skills: coding and understanding of algorithms, modeling, texturing, and eventually sound creation. Since time was very limited, texturing and sound creation has not been done in this project. Instead, some default textures from Quake 3 Arena and one default Windows sound have been used.

Interfacing with different hardware can be a (time-consuming) challenge if it does not come with a well documented SDK but only with incompatible modules.

The choice for the iterative development process proved itself right, unfortunately without any time to even run a first iteration. Fortunately the first version of the game was already usable for the experiments it was intended to. Also, the iterative process allowed for a certain degree of freedom in design while implementing, making it possible to quickly adapt to unknown properties or behavior of the OGRE framework.

After this first version of the game is finished, a lot of extra functionality and fixes could be implemented in future. In Appendix E, some recommendations for this can be found.
11. References


[Creative] Creative http://www.creative.com/


[InterSense] InterSense http://www.isense.com/products/pro/index.htm


[OGRE] OGRE 3D, April 2006 http://www.ogre3d.org/


[VTK] The Visualization ToolKit, April 2006 http://public.kitware.com/VTK/
Appendix A: Class diagrams

*Boat*

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mainApplication</td>
<td>Main application class</td>
</tr>
<tr>
<td>imageData</td>
<td>vtkImageData*</td>
</tr>
<tr>
<td>imageActor</td>
<td>vtkImageActor*</td>
</tr>
<tr>
<td>ren1</td>
<td>vtkRenderer*</td>
</tr>
<tr>
<td>renWin</td>
<td>vtkRenderWindow*</td>
</tr>
<tr>
<td>iren</td>
<td>vtkRenderWindowInteractor*</td>
</tr>
<tr>
<td>vector</td>
<td>vtkFloatArray*</td>
</tr>
<tr>
<td>escalar</td>
<td>vtkFloatArray*</td>
</tr>
<tr>
<td>tetraPoints</td>
<td>vtkPoints*</td>
</tr>
<tr>
<td>contourPlane</td>
<td>vtkContourFilter*</td>
</tr>
<tr>
<td>contourPlaneMapper</td>
<td>vtkPolyDataMapper*</td>
</tr>
<tr>
<td>contourPlaneActor</td>
<td>vtkActor*</td>
</tr>
<tr>
<td>hedge</td>
<td>vtkHedgeHog*</td>
</tr>
<tr>
<td>hedgeMapper</td>
<td>vtkPolyDataMapper*</td>
</tr>
<tr>
<td>hedgeActor</td>
<td>vtkActor</td>
</tr>
<tr>
<td>dataSet</td>
<td>vtkUnstructuredGrid*</td>
</tr>
<tr>
<td>PlanePolyData</td>
<td>vtkPolyData*</td>
</tr>
<tr>
<td>splatter</td>
<td>vtkGaussianSplatter*</td>
</tr>
<tr>
<td>planeFilter</td>
<td>vtkProbeFilter*</td>
</tr>
<tr>
<td>planeWidget</td>
<td>vtkPlaneWidget</td>
</tr>
<tr>
<td>lineActor</td>
<td>vtkActor</td>
</tr>
</tbody>
</table>

- main() |
- init() |
- keyEvent() |
- mainLoop() |
- cleanup() |
- vtkInit() |
- vtkRender() |
- vtkClose()
**Lungs**

```c
mainApplication

 imageData : vtkImageData*
nimageActor : vtkImageActor*
ren1 : vtkRenderer*
renWin : vtkRenderWindow*
iren : vtkRenderWindowInteractor*

mainViewer : CDataViewer*
MIPF : vtkVolumeRayCastMIPFunction*
lungMapper : vtkVolumeTextureMapper2D*
raycastMapper : vtkVolumeRayCastMapper*
lungOpacity : vtkPiecewiseFunction*
lungColor : vtkColorTransferFunction*
lungProperty : vtkVolumeProperty*
lungVolume : vtkVolume*

main()
init()
keyEvent()
mainLoop()
cleanup()

vtkInit()
vtkRenderWindow()
vtkClose()
```

**Oven**

```c
mainApplication

 imageData : vtkImageData*
nimageActor : vtkImageActor*
ren1 : vtkRenderer*
renWin : vtkRenderWindow*
iren : vtkRenderWindowInteractor*

lut : vtkLookupTable*
pts : vtkPoints*
ca : vtkCellArray*
pd : vtkPolyData*
scalar : vtkIntArray*
m : vtkPolyDataMapper*
a : vtkActor*

main()
init()
keyEvent()
mainLoop()
cleanup()

vtkInit()
vtkRenderWindow()
vtkClose()
```
mainApplication
- imageData : vtkImageData*
- imageActor : vtkImageActor*
- ren1 : vtkRenderer*
- renWin : vtkRenderWindow*
- ren : vtkRenderWindowInteractor*
- dipolos : Dipolos*
- dipolosText : Text*
- headmodel2 : headModel*
- dipoloscall : Dipoloscall*
- mo1 : charCallback
- lineActor : vtkActor*

read_dipolos
- readFile()
- getActor()
- getLegendActor()

headmodel
- getHeadModelActor()
- construct()
- getTransparentHeadModelActor()
- readFile()

Text
- setDipolosText()

callback
- Execute()
The grey classes are part of the OGRE Framework, classes with … have some methods or attributes omitted.
Appendix B: Setting up the software

This chapter covers all the settings needed to get the software running smoothly. First the Microsoft Visual Studio 2003 settings are described, then the VTK version settings and at last the settings for the HMD and stereoscopic vision. Note that the paths which have been used are for illustration purposes only and are different for each computer. They depend on where your software is installed.

Microsoft Visual Studio settings

In this project we worked with Visual Studio 2003. For the VR application, no extra configuration will be needed, since the solution files of the OGRE framework are well configured already. For the AR application, we will explain how to set the right paths for Visual Studio. When the solution file is opened in Visual Studio taskbar, left click on the boldfaced solution and choose the Properties option and a popup window will appear. You have to unfold the map labeled C++ and then select General. Here you see the option Additional Include Directories at the right side, where the VTK’s include directory must be added.

![Figure B.1: Add VTK to the additional include directories](image1)

Now unfold the map labeled Linker and choose for General again. This time the paths to the libraries of VTK and ARToolkit must be added to the Additional Library Directories. For example: C:\Program Files\vtk42\lib\vtk and C:\Projecto\Software\ARToolKit\lib.

![Figure B.2: Add VTK and ARToolkit to the libraries](image2)
There are some additional VTK libraries which have to be included in the at the Additional Options field, located in the Command line option instead of General. These are the following libraries:

- vtkCommon.lib
- vtkFiltering.lib
- vtkGraphics.lib
- vtkHybrid.lib
- vtkIO.lib
- vtkjpeg.lib
- vtkParallel.lib
- vtkPNG.lib
- vtkRendering.lib
- vtkZlib.lib
- libARvideo.lib
- libAR.lib
- libARd.lib
- libARgsub.lib
- libARgsub_lite.lib
- libARgsub_lited.lib
- libARgsub.Util.lib
- libAR.gsubUtil.lib
- libAR.gsubd.lib
- libAR.gsubUtild.lib
- libAR.gsubUtil.lib
- libAR.gsubd.lib
- libAR.video.lib

(Note: When using VTK 5.0, also vtkVolumeRendering.lib and vtkWidgets.lib needs to be included.)

When this is done, click OK.

Now go to the Tools option in the main menu of the Visual Studio taskbar and choose for Options. As before a new popup window with on the left some unfolded maps which can be chosen. This time unfold Projects and then choose for VC++ Directories. On the right appears a pull-down menu, labeled Show directories for. Here the same include and library paths must be added to Include files and Library files respectively: At the Include files the path C:\Projecto\Software\ARToolKit\include and C:\Program Files\vtk42\include\vtk must be added. At the Library files the path to the following include files must be added: C:\Projecto\Software\ARToolKit\lib and C:\Program Files\vtk42\lib\vtk.
Now the system path to VTK’s binaries must be set, so that Windows can find and use this. Therefore, right click on My computer, usually located at the desktop, and then choose for Properties. Now a popup screen appears, called System properties. Go to the Advanced tab and then go to the Environment variables button. Pushing this will show a new popup window, called Environment variables. In this window you see a box with system variables. If you scroll down you see a variable called path. At this moment it’s filled with the paths of Windows and other applications that are installed at that specific PC. If it is not yet included, then add the path where your VTK binaries are located. For example, this can be: C:\Program Files\vtk42\bin.

Setting the right VTK version
At IEETA not everyone is using the same version of VTK. Because these versions aren’t hundred percent compatible, we used with two different versions. The boat and brain application are created with VTK version 4.2 and the lung model with VTK version 5.0. Because these versions can’t operate simultaneously, you have to change all the paths to the VTK version you use. Don’t forget to change the path to VTK’s libraries in the Environment variables and restart your computer or log off the current user, otherwise these settings won’t be applied yet.

HMD and stereoscopic vision settings
To obtain stereoscopic vision some driver, application and hardware (HMD) settings needs to be changed.

To get vision at the HMD, you have to turn it on as first monitor in clone mode. For this you have to right click on your desktop and go to Properties. Now you see a popup window called Display properties. Go to the Settings tab. Now click on Advanced and a second popup window will appear.
In this popup window go to tab which has the name of your video card on it. In our case this is a 
*GeForce FX 5959 Ultra*, so the further explanation will only hold for those who have the same video 
card. When you click on this tab a new menu, with two already fold out options, will appear on the 
left. Go to *nView Display Settings* and switch the display mode to *clone* and make sure the i-Glasses is 
on top of the two displays (set as primary display). After this go to *Stereo properties* and set the 
desired *Stereo separation* (around 20% is enough for the AR application, around 60% is recommended 
for the VR game) and set the *Stereo type* to *DDC VGA glasses*. The other settings shouldn’t need any 
change.

The HMD itself also has to be set to accept stereo vision signals. This has to be done in the HMD 
menu, which can be entered by pushing de button on top of the HMD and hold it for a couple of 
seconds. On the HMD screen a menu with icons shows up. Go to the 3D icon and see what text is 
displayed. If it says “Auto – 3D1 – 3D2”, select and confirm the Auto option. If it says “Off – 3D1 – 
3D2”, select and confirm Off, go back into the menu, and reselect the Auto option. Since both the 
glasses and the video card support DDC signals, the glasses will automatically switch to 3D mode if 
the videocard sends out a 3D signal. All stereo settings are done now and you should be able to get a 
stereoscopic vision.
Appendix C: an example of a Maze game logfile

When the game finishes, a log file of this format will be saved in the directory \log.

This file contains the player events of the last game played. Format of this file: timestamp, eventname [, eventvalue]

=========

File name: log\demo.txt
Tracker: Off
Accelerometer: Off
Turning instead of strafing: Off

Total distance traveled: 1069 meters.
Game played for 5 minutes and 0 seconds.
Average speed of the user: 12.88 kilometers per hour.

3,target
3,collision
18,target
24,target
37,collision
42,collision
43,collision
46,fall,300
60,collision
63,target
68,collision
90,collision
93,target
96,target
102,target
106,collision
108,target
110,target
111,collision
114,collision
160,collision
164,collision
183,target
184,collision
188,collision
245,target
245,collision
251,target
264,target
267,collision
287,collision
296,collision

Total number of collisions: 18
Total number of targets found: 13 out of 21
Appendix D: edited files of the OGRE source

The OGRE source had to be edited and recompiled into a new Plugin_BSPSceneManager.dll and a new OgreMain.dll / OgreMain_d.dll file. To keep track of all changes and have this addition reproducible, all edited files have been noted here.

This edit was the most appropriate solution that could be implemented in a short time, simply enabling the application to get an EntityData structure from the SceneManager. In this EntityData only the name and viewpoint (position and orientation) are stored. Other properties that may have been set by a level editor, will not be passed on, since this will not be needed by this version of the game.

**File: ogrenew\OgreMain\include\OgreSceneManager.h**

*added under the struct ViewPoint definition:*

```cpp
/** structure for holding a viewpoint and entity name pair. */
struct EntityData {
    ViewPoint vp;
    String entName;

    EntityData(ViewPoint _vp, String _entName) {
        vp = _vp;
        entName = _entName;
    }
    EntityData(){}  // Constructor
};
```

*added under the virtual ViewPoint getSuggestedViewpoint definition:*

```cpp
virtual int getSuggestedViewpoint ( void ) { return 0; }
virtual EntityData getSuggestedViewpoint ( int entNum ) { return EntityData(); }  // Constructor
```

**File: ogrenew\PlugIns\BSPSceneManager\src\OgreBspSceneManager.cpp**

*added under the Viewpoint BspSceneManager::getSuggestedViewpoint definition:*

```cpp
int BspSceneManager::getSuggestedViewpoint ( void )
{
    return mLevel->mEntityList.size();
}
```

```cpp
EntityData BspSceneManager::getEntityData(int entNum) {
    return mLevel->mEntityList[entNum];
}
```

**File: ogrenew\PlugIns\BSPSceneManager\include\OgreBspLevel.h**

*added under the std::vector<ViewPoint> mPlayerStarts definition:*

```cpp
std::vector<EntityData> mEntityList;
```
added in the loadEntities method, under the bool isPlayerStart definition:

```cpp
bool isOtherEntity;
```

added under the isPlayerStart=false definition:

```cpp
isOtherEntity = false;
```

added under the if (params[0] == "classname" && params[1] == "info_player_deathmatch") { .... } statement:

```cpp
if (params[0] == "classname" && !isPlayerStart)
{
    isOtherEntity = true;
    entName = params[1]; //.c_str();
}
```

added under the if (isPlayerStart) { .... } statement:

```cpp
if (isOtherEntity)
{
    // Save other entity, only viewpoint data and classname will be stored!
    ViewPoint vp;
    vp.position = origin;
    vp.orientation.FromAngleAxis(angle, Vector3::UNIT_Z);
    EntityData ed;
    ed.vp = vp;
    ed.entName = entName;
    mEntityList.push_back(ed);
}
```
Appendix E: Recommendations

To improve the two applications, this appendix covers the recommendations that we think are useful for further research. First the recommendations of the AR application will be described and then the recommendations for the VR application.

**Augmented Reality application**

Because this application gives a better visualization of the models then the previous work methods we think many people will use our application from now on. However, our application has some features that we think can be improved. These improvements are related to the usability of the visualization, to run the program the cleaning of the memory.

**Usability**

Now a model is attached to the marker in a predefined position. Thus, the side of the model which is placed at this marker can’t be viewed from the bottom. A solution for this problem can be to place a different marker at the back of book, so when that marker is detected, the object will be shown from the bottom.

When the user wants to see a model in more detail, the only way to achieve this is to bring the marker closer to the camera. It appears to be that some people want to enter the model, so then the model must be scaled at a non predefined value. A solution might be to implement an external zoom function, which will improve the visualization of the model in more detail.

**Running of the program**

To start one of the application, the only way to do this is to open the solution file in Visual Studio and compile it. Of course there will be an executable generated, but for some reason this .exe file cannot be executed at all models. It would be an improvement for the user to be able to just run the executable, instead of recompiling the program.

The previous recommendation has also a consequence to the fact that there are now two different versions of VTK are used. That way some settings needs to be changed when the user wants to load another model. It is recommended for further development to change all the code to VTK version 5.0.

**Cleaning of the memory**

To clear the computer’s memory, all VTK members needs to be deleted at the end of the pipeline, before the application quits. Now the application will be exited by an internal function of ARToolKit, which results in the fact that the deletion of the VTK members will never be executed. This is no problem for the correct running and quitting of the application, but it would be better for the internal memory maintenance to free the space occupied by those VTK objects.

**Virtual Reality application**

The game intentionally was developed with an iterative development process. This means the first version of the application would be very simple, and not the final version. However, due to some unexpected time-consuming difficulty while implementing the collision detection, this idea did not work out in practice, so in this case, the final version is the first simple version without any iteration. Fortunately the game is useful for the experiments it was intended for, so not having walked through the iterative process was not a problem for this. When somebody might want to improve this game in the future, there are some recommendations of details that can be worked on.

**Collision detection**

As already said in the Implementation chapter, the collision detection that is being used right now, is just a proof of concept how to use RaySceneQueries to detect an object close to the player. In a next version, there should not only be checked for collisions in the walking direction, but everywhere
around the player. One of the most used techniques for collision detection in games is the use of an ellipsoid representing the player. With this ellipsoid or circle, you can determine all kind of collisions, sliding along walls, walking up stairs, and so on. This technique can be recommended and is described in an article by Paul Nettle [Nettle00]. Don’t forget the BSP SceneManager does not support clipping objects, so a solution to that problem might still be needed.

**User interface and configuration**
A little start has already been made by implementing parts of the CEGUI interface tool of OGRE. When continuing development at this game, making a GUI for configuration would be much better than letting the researcher edit a configuration file each time before the game can be started.

**Audio engine**
At this moment, the game can only play one sound when a target is found. This is hardcoded in the Obj_Box.cpp file, but should be handled by a separate sound engine. Using a separate sound engine would also allow for some background music and other sound effects.

**Accelerometer**
The implementation of the accelerometer has some bugs, randomly stalling the application. This probably has something to do with a thread not locking correctly, but the precise issue has not been found yet. If eventually someone would like to run the game with the accelerometer instead of the solution with the mousebuttons, it is recommended to fix this bug to make the application run stable with the accelerometer.

**Tracker**
The tracker implementation now requires the user to keep his head straight until the application has been fully loaded. A try to implement a software reset of the tracker did not work out so far, but it would be a great improvement if this eventually would work.

**Mouse**
When the mouse is moved very quickly, the “prevent the camera from pitching too hight or low”-code does not stop the camera from moving. This results in the player getting an upside down view, from which he is unable to recover. Although this happens only very occasionally, this is an issue which should be taken care of.

**Memoryleaks**
The OGRE framework reports memoryleaks automatically in the file “OgreLeaks.log”. In this file, we will be congratulated for not having any memory leak, but when we quit the application inside the Visual Studio debug mode, the output window clearly shows a lot of memory leaks after exiting the game. This leak appeared since the SerialSensor code has been added. Fortunately, the memory usage of the game does not grow, so this is not a big issue.

**Level**
The level in this state is not a very complicated model: just some walls and a ceiling, some textures on it, and a room is finished. Adding extras to the level and using custom textures instead of Quake 3 Arena textures, would give the game a more appealing view.

**Extra functionality**
Finally, when the basic game is running smoothly, extra functionality could be added. This could be anything, from new levels and objectives to dynamic enemies, just be creative, that’s all you need when developing a game!