Entwicklung eines Augmented-Reality-Spiels

Studienarbeit
im Studiengang Computervisualistik

vorgelegt von
Jan Heitger

Betreuer: Prof. Dr.-Ing. Stefan Müller
Institut für Computervisualistik, AG Computergraphik

Koblenz, im März 2010
Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ja  Nein

Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden. □  □

Der Veröffentlichung dieser Arbeit im Internet stimme ich zu. □  □

(Ort, Datum)  (Unterschrift)

Die Studienarbeit hat das Ziel diesen sehr interessanten Teil der Computergrafik nicht nur sichtbar sondern erfahrbar zu machen. Zu diesem Zweck soll eine Anwendung erstellt werden, die eine einfache und direkte Interaktion mit augmentierten Elementen der Szene ermöglicht und die Möglichkeiten, die eine Augmented Reality Umgebung bietet, in Form eines Spiels erfahrbar macht.

Die Herausforderungen der Arbeit sind hauptsächlich in der Integrierung der verschiedenen involvierten Komponenten zu sehen, sowie in der Anpassung des Spiels an die Besonderheiten der Augmented Reality.

# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Project Idea</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Goal</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Structure</td>
<td>2</td>
</tr>
<tr>
<td><strong>2 Basics</strong></td>
<td>4</td>
</tr>
<tr>
<td>2.1 Augmented Reality and Tracking</td>
<td>4</td>
</tr>
<tr>
<td>2.2 ARToolkit</td>
<td>5</td>
</tr>
<tr>
<td>2.2.1 Functionality</td>
<td>6</td>
</tr>
<tr>
<td>2.2.2 Structure</td>
<td>6</td>
</tr>
<tr>
<td>2.2.3 Properties</td>
<td>8</td>
</tr>
<tr>
<td>2.3 ARToolkit Plus</td>
<td>8</td>
</tr>
<tr>
<td>2.3.1 Functionality</td>
<td>8</td>
</tr>
<tr>
<td>2.3.2 Structure</td>
<td>8</td>
</tr>
<tr>
<td>2.3.3 Properties</td>
<td>9</td>
</tr>
<tr>
<td>2.4 Work of M. Brümmer</td>
<td>10</td>
</tr>
<tr>
<td>2.4.1 Structure</td>
<td>10</td>
</tr>
<tr>
<td>2.4.2 Properties</td>
<td>10</td>
</tr>
<tr>
<td>2.5 Ogre 3D</td>
<td>10</td>
</tr>
<tr>
<td>2.5.1 Advanced Ogre Framework</td>
<td>11</td>
</tr>
<tr>
<td>2.5.2 WebCamPlugin</td>
<td>11</td>
</tr>
<tr>
<td>2.5.3 Ogre ARToolkit(Plus) Integration</td>
<td>11</td>
</tr>
<tr>
<td>2.6 Game concept</td>
<td>11</td>
</tr>
<tr>
<td>2.6.1 Modification to usual Genre Structure</td>
<td>14</td>
</tr>
<tr>
<td><strong>3 Implementation</strong></td>
<td>16</td>
</tr>
<tr>
<td>3.1 WebcamPlugin</td>
<td>16</td>
</tr>
<tr>
<td>3.2 ARToolkitPlus</td>
<td>18</td>
</tr>
<tr>
<td>3.3 PTypes dll</td>
<td>19</td>
</tr>
<tr>
<td>3.4 Advanced Ogre Framework</td>
<td>19</td>
</tr>
<tr>
<td>3.4.1 Tracking/Integration of M. Brümmer’s work</td>
<td>20</td>
</tr>
<tr>
<td>3.5 GameState</td>
<td>22</td>
</tr>
<tr>
<td>3.5.1 Derived methods</td>
<td>22</td>
</tr>
<tr>
<td>3.5.2 handleTowers()</td>
<td>24</td>
</tr>
<tr>
<td>3.5.3 handleCpu()</td>
<td>26</td>
</tr>
<tr>
<td>3.5.4 handleViruses()</td>
<td>28</td>
</tr>
<tr>
<td>3.5.5 handleProjectiles()</td>
<td>30</td>
</tr>
<tr>
<td><strong>4 Results</strong></td>
<td>32</td>
</tr>
<tr>
<td>4.1 Installation</td>
<td>32</td>
</tr>
<tr>
<td>4.2 Gameplay</td>
<td>32</td>
</tr>
</tbody>
</table>
1 Introduction

Augmented reality is a part of computer graphics, which focuses on displaying virtual images on top of real world scenes. The use of augmented reality is not yet as widespread as the use of virtual reality, which encompasses the majority of computer graphics applications, but the technology has great potential for a lot of applications, ranging from medicine and construction to design and entertainment. The idea behind this work is to create an application, which allows the users to experience the potential of augmented reality in a playful way.

Augmented reality is a complex part of computer graphics dealing with a variety of problems ranging from tracking of real world objects to rendering with the limited hardware capabilities of mobile devices. Part of this work is the familiarization with the various parts needed for an augmented reality application; from acquiring the needed real world information, to rendering the virtual images using a computer graphics engine.

1.1 Project Idea

To allow the users to experience augmented reality in an interactive and fun way, the application takes the form of a game, in which the interaction with the augmented scene elements is part of the gameplay. There have been a few games which feature an augmented reality environment with varying degrees of interactivity. Among these are big commercial releases from Sony Entertainment like Eye of Judgment and EyePet, as well as small game projects from independent developers.

![Figure 1: The Eye of Judgment, Sony Computer Entertainment [1]](image1.png)
One project that served as inspiration for this work is levelHead by Julian Oliver, which features a very direct interaction with the augmented elements.

![Figure 2: levelHead, Julian Oliver [2]](image)

1.2 Goal

The goal of this work is to create an augmented reality game that familiarizes the player with the concept of augmented reality and provides meaningful interaction with the game elements. To achieve a high accessibility the application will require no special hardware like a HMD (head mounted display) or a specific camera and will be developed under windows. Further requirements are an appealing presentation with clear and easily understandable graphics and animations, beginner-friendly gameplay that provides interesting interaction mechanics with the augmented scene elements.

1.3 Structure

The second chapter of this work deals with the basics necessary for this work. Augmented reality and the involved tracking methods are explained, with a focus on optical tracking. The chapter also discusses the used components from the ARToolkitPlus, which provides optical marker tracking, to the Ogre 3D engine and the associated plug-ins used for rendering. Additionally the game design is explained and the special gameplay requirements for an augmented reality game are detailed.
Chapter three deals with the integration of the involved components and shows the implementation of the game, by giving an overview of the architecture and important methods. At the end of the work the results are shown and potential ways to continue the project are given in a conclusion.
2 Basics

2.1 Augmented Reality and Tracking

Augmented Reality is a part of computer graphics which deals with combining real world visuals with computer generated images, thus creating a mixed reality environment. Augmented Reality is part of the Mixed Reality Continuum formulated by P. Milgram and F. Kishino [3], which encompasses the spectrum from the real world to virtual reality.

![Mixed Reality Continuum](image)

**Figure 3:** Mixed Reality Continuum [3]

Between these two extremes Mixed Reality defines a space in which virtual scenes are combined with real world information (Augmented Virtuality) or in which real world scenes are enriched by virtual information (Augmented Reality). Augmented Reality is already used in a multitude of applications, ranging from sports TV to construction and medicine, where information is presented as overlays over real world objects to guide the user. The added information is presented in different forms, e.g. as virtual object or text.

![Wikitude application](image)

**Figure 4:** Wikitude application [4]

To ensure the proper positioning of the contextual information a tracking system is used. These systems operate with different sensors to determine positions, e.g. mechanical tracking, ultrasound, GPS or optical tracking.
Many Augmented Reality systems make use of optical tracking of which there are two distinct variants. Out-side-in tracking uses external cameras in combination with active or passive markers that are placed on the user, to calculate position and orientation. It is often used for motion-capturing or body tracking. In-side-out tracking places the camera on the user and searches for fiducial markers to determine position and orientation of the objects or the user.

Marker-based optical tracking is limited in that the markers have to be clearly visible at all times to provide exact tracking but has the advantage that the markers can be easily printed. There is also marker-less optical tracking which forgoes the need for markers by directly tracking objects. These objects are usually defined beforehand so the system can identify them.

Marker-based optical tracking was chosen for this work, because of the exact position and orientation provided and the ease of use of the markers. (e.g. users can easily print their own markers)

2.2 ARToolkit

ARToolkit is the most prominent AR-Tracking system. It is available as an open source library for non-commercial use and includes an Open-GL based graphic output system, making it an ”Out-of-the-box” AR system. Figure 4 shows an example of the tracking and rendering ARToolkit provides.

![ARToolkit SimpleTest](image)

Figure 5: ARToolkit, SimpleTest [5]

A single Marker is tracked and a cube is displayed on top of it in regards to the marker’s position and orientation. ARToolkit is platform-independent
and can be compiled under Windows, Linux and MacOS. It was written in C and provides no class bases API.[5]

2.2.1 Functionality

To provide its functionality ARToolkit uses a system in which the camera input is processed to determine the marker’s position. First the image supplied by the camera is converted into binary space, which makes it easier to find the quadratic, black marker borders. With this information the position of the marker corresponding to the camera is calculated. Next the marker-id, the symbol inside the marker border, is identified and matched against the library of trained ids. Afterwards the virtual object that is bound to the detected marker-id is rendered on top of the camera image using the calculated position and orientation values, thus resulting in an augmented image. This is done for every frame. Figure 5 shows an overview of the the ARToolkit tracking process.

![Figure 6: ARToolkit, Functionality][5]

2.2.2 Structure

The ARToolkit architecture builds on top of OpenGL, the system Standard API and its Video library. GLUT is used for window/event management. ARToolkit itself is separated in three parts:

- The video module, which builds on top of the systems video library and is responsible for video input and processing.
- The core module, which handles the marker tracking routines.
- The Gsub module, or in newer versions, the Gsub-lite module which handles graphic output.
These modules operate in a pipeline model, which makes it possible to exchange parts of it. (e.g. change to a different video input module)
2.2.3 Properties

ARToolkit provides marker-based optical tracking and an Open-GL based rendering module. There is a predefined set of markers available for use and the user can create additional markers, which have to follow the same structure as the others (white plane with quadratic, black borders) The symbol inside the borders can be customized using an image editing program. Using simple asymmetric symbols achieves better tracking results. To register the new markers, ARToolkit provides a program which allows the user to generate the necessary files, which will then be used in the template matching process. Multi-markers are another option for tracking with ARToolkit. These markers consist of several single markers whose relative position is known to ARToolkit. This way the program can determine the position of the multi-marker even if some parts of it are occluded.

2.3 ARToolkit Plus

ARToolkitPlus is a successor to the original ARToolkit. It improves on many features of its predecessor, among others improved tracking algorithms, improved performance, id-encoded markers and a C++ class based API.

2.3.1 Functionality

2.3.2 Structure

There are several differences in system architecture between ARToolkitPlus and its predecessor. The most notably is probably the change from a C-based, complete Augmented Reality application system that incorporates rendering capabilities, to a C++ based pure marker tracking library. That means the Gsub and the video module are no longer part of the system and the user is required to supply his own video input and rendering programs. The difference in the core tracking module is also of note as it was changed to a C++ class based API. The most important classes from a developer’s perspective are:

- Class Tracker: This is the main class of ARToolkitPlus and holds all functions of the core module of the original ARToolkit.
- Class TrackerMultiMarker, TrackerSingleMarker: These classes are derived from Tracker and hold the functionality for the use of single- and multi-marker tracking, expanding on the parent class.

The composition of the involved classes is rather complex. Figure 10 shows an overview of the tracker classes, which are implemented in their respective header files.
Tracker, TrackerSingleMarker and TrackerMultiMarker are abstract classes and are used as interfaces for the derived classes. They are not meant to be included into a class themselves, instead the implementation classes of the respective tracker type is used. For example the class TrackerSingleMarkerImpl is derived from both TrackerSingleMarker and Tracker and, as it is not an abstract class, implements all of their methods. These classes are then used to create the instances for use in an application.

### 2.3.3 Properties

ARToolkitPlus provides marker-based optical tracking, just like the original ARToolkit. The marker identification is no longer achieved by template matching though. Instead ARToolkitPlus uses id-encoded markers. The symbol inside the marker borders is replaced with a binary code image, which is decoded to determine the marker-id. Depending on the type of marker selected there are 512(simple) or 4096(bch) available ids.

Another feature of the ARToolkitPlus is automatic brightness thresholding. It improves upon the marker detection of the original in low or bright light situations even if there are changes in lighting conditions during runtime. ARToolkitPlus is also suited for use with mobile devices as it includes a memory management system, making it easier to work with the hardware’s limitations.
2.4 Work of M. Brümmer

In her work [6] Martina Brümmer created a complete marker tracking application based on ARToolkitPlus, adding video input and rendering capabilities via OpenGL and implemented additional classes to improve tracking of multiple single markers.

2.4.1 Structure

For video input Martina Brümmer chose to add the video library of the original ARToolkit. For this the libARVideo library was extracted and merged with ARToolkitPlus. Two additional classes were introduced:

- Class Image contains relevant image information, e.g. width, height, pixel format and methods to generate a gray scale image.
- The class CamVideo manages the video input and is responsible for opening and closing of the video stream.

For ARToolkitPlus two new wrapper classes were added, that deal with marker tracking:

- The class TrackCoord contains the relevant information for a detected marker e.g. his PatternId, World- and Eye-Matrix.
- The class TrackerMultiSingleMarkerImpl is a wrapper around TrackerSingleMarkerImpl and improves upon the tracking algorithm for detection of multiple single markers by correlating their position data.

2.4.2 Properties

Martina Brümmer’s work provides an example for a complete ARToolkitPlus application and improves upon the single marker tracking found in the original ARToolkitPlus.

2.5 Ogre 3D

Ogre3D is an open source 3D graphics engine that runs on a variety of operating systems. Its structure is object-oriented and there is a wide variety of plug-ins to add functionality. Ogre was written in C++ but there are some ports for different languages. Ogre supports multiple processors and can be used with DirectX and well as OpenGL, as well as the shading languages Cg, HLSL and GLSL. The open source community of Ogre provides documentation (in form of the Ogre3D Wiki) and is actively involved in the further development of the engine. Ogre is available under the Open Source License(LGPL) or under the MIT License, starting from version 1.7. This work uses Ogre version 1.6.4. The following details the plug-ins that were used in this work.
2.5.1 Advanced Ogre Framework

The Advanced Ogre Framework is a continuation of the Basic Ogre Framework and provides a starting point for Ogre applications with a custom built framework, instead of the Example Framework which is used in the Ogre tutorials. The 14 classes of the framework provide a variety of features such as:

- A game state system to switch between different environments in the application. Examples for the environments are different levels in the game, as well as menu or intro screens, which all have specific requirements that can be modeled in their respective state class.

- A graphical user interface which is based on the CEGUI library used by Ogre.

- An input manager which comes with support for keyboard and mouse input, but can be expanded to use more input systems.

The relevant classes will be detailed in the implementation part of this documentation. Documentation can be found in the Ogre Wiki [8].

2.5.2 WebCamPlugin

A Webcam Plugin for Ogre written by the user Timme that makes it possible to register a webcam image stream as a texture source in Ogre. Changes made to the plug-in will be explained in the implementation chapter.

2.5.3 Ogre ARToolkit(Plus) Integration

A library that is the product of an integration of ARToolkitPlus into the Ogre engine, written by the user Futnuh [9]. It was used to write the ARToolkitPlus implementation for this work. The resulting classes will be detailed in the implementation chapter.

2.6 Game concept

The idea for this work is to create a game that allows the user to experience Augmented Reality in a direct and immediate way. The user should be able to not only see virtual objects as overlay to his normal vision, but to interact with them in a way that feels natural and meaningful. That means the game has to have a meaningful interaction mechanic that consists of more than a remapping of mouse and keyboard input, a problem that often arises with the new input devices found in modern game consoles, such as the Wii or the Playstation 3 (the Wii Remote or the SixAxis Controller for the Playstation 3).
Many games for these systems add additional control schemes in an effort to utilize the new input devices to enhance the gameplay, often disregarding the fact that a traditional control scheme works better for the game. Games whose gameplay is actually enhanced by the new input mechanics are often built specifically around the strengths and limitations of the input devices.

For this work the gameplay should focus around the use of Augmented Reality, or more precisely the interaction with markers used for optical tracking. Additional requirements are that the user should be able to play the game while sitting in front of the pc and the scope of the game must be simple enough to be completed in the timeframe of this work.

As the game is supposed to pique people’s interest in the technology, it needs to be beginner-friendly with transparent game mechanics. The hardware requirements should be low to allow a wide range of users to play the game. This refers to system specifications like processor speed as well as the setup itself, e.g. the setup does not involve a HMD only a standard TFT display.

The chosen interaction with markers using optical tracking imposes certain restraints on the gameplay. Optical tracking is very precise but most tracking systems have problems with detecting moving markers, which makes it unsuitable for most action game concepts. Another problem is the fact that the detection is severely hindered by marker occlusion. ARToolkit and ARToolkitPlus cannot detect markers when the marker or the marker border is partially occluded. If the user is supposed to interact with markers as a regular gameplay mechanic this limitation needs to be accounted for.

With the given requirements a game of the strategy genre seems most suited, as it evades the technical problems that would arise in a fast paced action game. The game needs to be easily accessible and feature a clear and simple game mechanic. Suitable members of the strategy genre are the tower defense style games.

In these games the player is usually tasked with the defense of a structure against waves of enemies that advance on predefined paths towards the structure. Their attack is countered by building the name-giving towers, defensive structures that shoot or otherwise engage the enemies. The different types of towers have individual strengths and weaknesses, so the challenge lies in finding the right combination of towers to counter the waves of enemies. Usually the player is limited in how many towers he can build by some kind of score-based resource and can construct additional towers the longer the game progresses, resulting in more complex tower formations over time. Figures 11 and 12 show examples of a tower defense game. Notable entries in the genre are Desktop Tower Defense, Plants vs. Zombies or Defense Grid: The Awakening.
Figure 11: Desktop Tower Defense [10]

Figure 12: Plants vs Zombies [11]
2.6.1 Modification to usual Genre Structure

A tower defense game that takes place in an Augmented Reality environment is not limited to the computer screen; instead the playing field is the literal desktop in front of the player. Ideally the player would wear some kind of HMD to see the virtual overlays but the requirements state that only a normal pc display is available. In this case the user sees the desktop on the computer screen, as captured by a video camera, augmented with overlaid information and manipulates the markers that lie in front of him according to the information presented on the screen. To enhance usability the displayed information needs to be clearly visible. The user is supposed to look at the screen and be able to orient himself this way. For this, the screen needs to be relatively free to allow the player to see as much of the desktop image as possible. As a consequence there are no graphical user interface elements in the game, like a score or resource window. Instead all relevant information is given to the player in the form of a visual indicator. E.g. the hitpoints left are represented by a change in color on the central structure. Additionally the number of towers the player constructs, should be relatively small, so they don’t occlude too much of the screen.

A normal tower defense game requires little input from the player. Mouse input is the only thing needed to select and place the towers and this usually encompasses the control mechanics of a tower defense game. For this Augmented Reality tower defense game the construction of the towers is done via placement of the markers. As soon as they are placed on the playing-field the markers are detected and the towers are constructed. As the number of towers the player constructs, should remain small, another deviation from the usual tower defense gameplay is needed, where the player constantly builds new towers and upgrades existing ones. To address this, a new gameplay mechanic is introduced in the form of a resource pool that empties for every tower that is currently in the playing-field and that is refilled for every tower removed. Additionally the waves of enemies will advance from four different directions. This combined with the limited amount of towers the player can build, encourages the player to relocate existing towers, keeping the total number of towers in the game small.

To make the interaction with the markers meaningful and to offer a unique interaction metaphor, the towers are linked to the markers. If this were not the case the only interaction with the virtual items would be the construction process itself. The player could place a marker on the playing-field to construct a tower, but would then lose any ability to interact with the virtual object he just created. By linking the markers with the towers, they stay in a comprehensible relationship, giving the player the feeling that he directly interacts with the virtual objects by touching and moving the markers. Accordingly a tower is constructed by placing a marker on the
playing-field and moved to a different location by moving the marker to that location.

To make tower placement an interesting strategic choice, the process for construction and deconstruction of a tower takes a certain amount of time. Once the marker is placed, the tower enters a construction phase until it is finished and will equally enter a deconstruction phase once the marker is removed.

This also helps to address the problem of marker occlusion. When a marker for a given tower is occluded for a short amount of time e.g. by the arm of the user as he moves a marker to another place, the tower will just enter its deconstruction phase for a short amount of time and will revert back to its finished form in just the same amount of time it was occluded.
3 Implementation

This chapter details the implementation part of the game. Microsoft Visual Studio 2008 was used for development and the following will illustrate the projects that are found in the solution.

Ogre uses two main classes to manage its scene objects, Entity and SceneNode. Ogre::Entity is a representation of an object’s geometry, managing the object’s mesh and sub meshes. Creating an Entity alone is not sufficient to create an object in a scene though, as it also needs to be inserted into an active scene graph. This is accomplished by attaching an entity to a scene node.

An entity also holds the animation states that are linked to the mesh. To animate an object the animation state is activated and then advanced by adding time to it.

mAnimationState = mEntity->getAnimationState(animationState);
mAnimationState->setLoop(true);
mAnimationState->setEnabled(true);
mAnimationState->addTime(time*0.001);

The abstract class RenderableObject is a wrapper around the Ogres Node and Entity classes, providing basic management methods like get/set for the object’s orientation, position and animation state, as well as a basic draw method. The child classes implement additional functionality for their respective objects. E.g. the abstract Tower class specifies additional methods dealing with target acquisition and resource management. The class hierarchy ends with classes for the specific game objects, such as VectorTower, a specific tower object, or Grunt, a specific Virus. Figure 13 shows a class diagram of the RenderableObject hierarchy.

3.1 WebcamPlugin

This project contains the webcam plug-in written by the user Timme of the Ogre community [7]. It uses OpenCV to manage Video input in the Ogre environment and creates a dll for use in the main application and the PTypes library for multithreading. The dll created must be present in the directory of the main application and the library file must be linked against the main application’s project. It contains four classes:

- The class WebcamController is the management class for the plug-in and an instance of this class has to be created in the main application to initialize the webcam. It takes care of initializing the plug-in and creates the texture, which is used to display the webcam image stream.

- The class WebcamTexture is used to manage and update the texture created by the plug-in. It also contains the transformation matrixes
Figure 13: RenderableObject hierarchy
that are used to manipulate the texture, e.g. rotate it, in case the web-
cam image needs to be flipped around an axis for correct alignment.

- The main part of the functionality is found in the Webcam class, where
  the image is created and converted into the proper format for display-
ing, using OpenCV.

- The abstract class WebcamListener is used to register a listener in
  Ogre and to update the texture for every new frame.

The code used to initialize the plug-in in the main application:

```cpp
ExternalTextureSourceManager::getSingleton().setCurrentPlugIn("webcam_video");
WebcamController* WebcamCtrl = (WebcamController*)
  ExternalTextureSourceManager::getSingleton().getCurrentPlugIn();
WebcamCtrl->getDictionaryStringName();

mWebcamListener = new ARWebcamListener(m_pCamera, WebcamCtrl->
  getWebcamByNumber(0));
WebcamCtrl->getWebcamByNumber(0)->addWebcamListener
(mWebcamListener);
```

For the use of the plug-in a specific input in the material, used to display
the webcam stream, is required to set the texture source and to configure
some options for the plug-in:

```cpp
material <MaterialName>{
  technique{
    pass{
      texture_unit{
        texture_source webcam_video{
          //Flip true //flips around the y axis
          //Webcam_Number 0 //Defines which webcam to use
        }
      }
    }
  }
}
```

### 3.2 ARToolkitPlus

This project contains the ARToolkitPlus code. It compiles a static library
which is used in the main application. No changes were made to this
part. The added code from Martina Brümmer’s work can be found in the
AdvancedOgreFrameWork project as TrackCoord and TrackerMultiSingle-
Marker classes.
3.3 PTypes dll

This project contains the code for PTypes library used by the webcam plugin. No changes were made to the code. It compiles a dll file, which must be present in the directory of the main application and a lib file for the webcam plugin.

3.4 Advanced Ogre Framework

The Advanced Ogre Framework, as described in chapter 2 is a framework for Ogre Applications that includes features that go beyond the example framework used in the Ogre Tutorials. It holds the main application for this work and its most important classes will be described here. The Ogre Wiki entry holds a detailed documentation [8].

The OgreFramework class forms the basis for the framework and contains most of the Ogre related variables.

- The abstract AppstateListener class defines functions to be inherited later by an instance of this class that deal with the management of the state system. The AppState objects are stored in a std::Vector in the derived class. The active State is on top of the Stack and can be changed/removed with the functions.

- The abstract class AppState acts as a blueprint for the states and defines functions for them to inherit. The class provides functions to enter, exit, pause and resume an AppState, as well as the update function that is called every frame. Every AppState also contains pointers to the Ogre sceneManager and camera objects.

- AppStateManager inherits from the abstract class AppstateListener and implements its functions. As mentioned the class also contains vectors to store AppState objects. A vector to store all existing AppState objects and another one for the active states.

- MenuState is an implementation of the AppState class and contains all functionality needed for a menu screen with two buttons. In addition to the management functions of the AppState class (enter(), exit(), etc.) it contains input handling functions for mouse and keyboard input.

- GameState is another implementation of AppState and contains an example for an interactive scene in the standard Advanced Ogre Framework. In this work the GameState class forms the basis for the tower defense game and will later be explained in greater detail.
DemoApp class is the most top level class in the framework and contains an instance of the AppStateManager. It is called from the program's main() and starts the actual application.

Figure 14 shows a class diagram of the structure used by the state system of the Advanced Ogre Framework.

![Class Diagram](image)

**Figure 14:** Class diagram of the state system

### 3.4.1 Tracking/Integration of M. Brümmer’s work

The central class for tracking functionality is ARWebcamListener which can be found in the ARListener.h file. It inherits from the WebcamListener class of the Webcam Plugin and uses the modified ARToolkitPlus functions for tracking. An instance of the class is created in GameState and linked with the Webcam by the WebcamController.

```cpp
mWebcamListener = new ARWebcamListener(m_pCamera, WebcamCtrl->getWebcamByNumber(0));
```
OgreFramework::getSingletonPtr()->m_pLog->logMessage
("Adding WebcamListener");
WebcamCtrl->getWebcamByNumber(0)->addWebcamListener
(mWebcamListener);

The ARWebcamListener contains a member variable for an OpenCV image that gets supplied by the webcam.

mImage = webcam->getCvImage();

It also contains a pointer to an instance of the TrackerMultiSingleMarker-Impl class that was written by Martina Brümmer and contains the modified tracking algorithm for multiple single markers. It is created with:

mTracker = new CamVid::TrackerMultiSingleMarkerImpl
    (mImage->width,mImage->height);

The tracker is then configured with the marker size in mm and a camera calibration file.

mTracker->initSettings(80, "camera_para.dat");

Another member variable contains the projection matrix with is supplied by ARToolkitPlus and is used in the GameState class to align the coordinate system.

float* mp = (float*) (mTracker->getProjectionMatrix());
mProjectionMatrix =
    Ogre::Matrix4( *(mp+0), *(mp+4), *(mp+8), *(mp+12),
    *(mp+1), *(mp+5), *(mp+9), *(mp+13),
    *(mp+2), *(mp+6), *(mp+10), *(mp+14),
    *(mp+3), *(mp+7), *(mp+11), *(mp+15));

Ogre uses the same notation for Matrixes as OpenGL. That means a 4x4 Matrix consist of four single column, 4-row matrixes that represent the vectors. Ogre always uses this internal notation, regardless if the DirectX or OpenGL rendering mode is selected. For Direct X the matrix orientation is changed internally.

The function newFrame() is called by the webcam plug-in in the class Webcam and contains the call for the actual tracking algorithm. The results are saved in mMarkers.

mMarkers = mTracker->calcMarkers((unsigned char*)
    (mImage->imageData),isWorldMarkerCached);

mMarkers is a vector of TrackCoord objects and is the actual result of the tracking process that is used in the GameState class.

Instances of the TrackCoord class are created for every detected marker and contain information relevant for that specific marker. Among the information calculated are the markers position and orientation(in Ogre::Vector3
and Ogre::Quaternion format), its modelview matrix and PatternID. It also contains a boolean variable to identify if the marker has an associated tower, which is set later in the GameState. Changes made to Martina Brümmer’s code are game specific additions (position and orientation in Ogre formats, etc.)

3.5 GameState

The GameState is the central class for the tower defense game and is an implementation of the abstract AppState class. Figure 15 shows a class diagram of the GameState class and shows the use of the connected components.

![Class diagram of the GameState class and its connected components](image)

Figure 15: Class diagram of the GameState class and its connected components

3.5.1 Derived methods

The derived methods deal with the management of the state. The enter() and update() methods have been modified for the game. In enter() the Ogre
environment is configured. The SceneManager is created:

```cpp
m_pSceneMgr = OgreFramework::getSingletonPtr()->m_pRoot->
    createSceneManager(ST_GENERIC, "GameSceneMgr");
```

The camera is created and placed, clipping distance and aspect ratio are set.

```cpp
m_pCamera = m_pSceneMgr->createCamera("GameCamera");
m_pCamera->setPosition(Vector3(0,0,500));
m_pCamera->lookAt(Vector3(0,0,-300));
m_pCamera->setNearClipDistance(5);
m_pCamera->setAspectRatio(Real(OgreFramework::getSingletonPtr()->m_pViewport->getActualWidth()) /
    Real(OgreFramework::getSingletonPtr()->m_pViewport->getActualHeight()));
OgreFramework::getSingletonPtr()->m_pViewport->
    setCamera(m_pCamera);
```

The Webcam plug-in is called and the ARWebcamListener is created and registered

```cpp
ExternalTextureSourceManager::getSingleton().
    setCurrentPlugIn("webcam_video");
WebcamController* WebcamCtrl = (WebcamController*)
    ExternalTextureSourceManager::getSingleton().
    getCurrentPlugIn();
WebcamCtrl->getDictionaryStringName();
mWebcamListener = new ARWebcamListener(m_pCamera,
    WebcamCtrl->getWebcamByNumber(0));
OgreFramework::getSingletonPtr()->m_pLog->logMessage
    ("Adding WebcamListener");
WebcamCtrl->getWebcamByNumber(0)->addWebcamListener
    (mWebcamListener);
```

The projection matrix supplied by ARToolkitPlus via the ARWebcamListener is applied:

```cpp
m_pCamera->setCustomProjectionMatrix(true,mWebcamListener
    ->getProjectionMatrix());
```

and the files pertaining to the enemy formations are read. The creation and handling of the enemies will be explained later in the chapter.

```cpp
readVirusFile("North");
readVirusFile("West");
readVirusFile("South");
readVirusFile("East");
```

Next the createScene() method is called. There the rectangle that is used to display the webcam image is created and registered with the SceneManager.
mRect = new Rectangle2D(true);
mRect->setCorners(-1.0, 1.0, 1.0, -1.0);

mRect->setBoundingBox(AxisAlignedBox(-100000.0*Vector3::UNIT_SCALE, 100000.0*Vector3::UNIT_SCALE));
mRect->setMaterial("ARRectangle");
mRect->setRenderQueueGroup(RENDER_QUEUE_BACKGROUND);
node = m_pSceneMgr->getRootSceneNode()->createChildSceneNode("Background");
node->attachObject(mRect);

The used material script ARRectangle contains the texture source information to display the webcam image and is set to receive no light from external sources and is not written into the depth buffer, resulting in other objects being drawn on top of it. The material script for ARRectangle:

```
material ARRectangle{
    technique {
        pass{
            lighting off
            depth_write off
            texture_unit{
                texture_source webcam_video{
                    //Flip true
                }
            }
        }
    }
}
```

The ambient scene lighting is setup with

```
m_pSceneMgr->setAmbientLight(Ogre::ColourValue(0.7, 0.7, 0.7));
```

After this the update() method is called for every new frame, which contains the handling methods for the game entities: Towers, Cpu, Virus and Projectiles.

### 3.5.2 handleTowers()

As mentioned earlier the tower objects in the game and the markers are linked to each other. That means when a marker is detected, a tower object is created and enters the construction phase. During this phase the resource available to the player, responsible for limiting the number of towers simultaneously in the game (named ClockCycles) is diminished accordingly to the cost of the tower. Additionally the tower objects animation phase is advanced to visualize the construction. If the marker beneath the tower is removed or the detection stops due to occlusion, the construction process is reversed, ClockCycles are added back the resource pool and the animation plays backwards. The tower object keeps a reference to its completion
status and enters an active state if it reaches full construction status or is
destroyed when its construction is completely reversed. If a marker of the
same kind (of the same PatternId) as the original marker is placed beneath
a deconstructing tower or the occlusion that resulted in the deconstruction
process stops, the construction process will continue.
For each marker detected a TrackCoord object is created and used in GameS-
tate via a Vector of TrackCoord objects. The tower objects are stored in a
Vector as well, called mTowers. For the construction of a tower several
variables are evaluated:

- hasTower() in the TrackCoord object checks if the marker is already
  linked to a tower object,
- getPattId() retrieves the markerId, which is used to determine the
  type of tower to create
- isWithinLanes() in GameState checks if the marker is placed on an
  invalid part of the playing-field

```cpp
if (!mMarkers[i].hasTower() && mMarkers[i].getPattId() != -1 &&
    mMarkers[i].getPattId() != 0 && !isWithinLanes(mMarkers[i])){
    constructTower(mMarkers[i].getMarkerPosition(), mMarkers[i].
        getOrientation(), mMarkers[i].getPattId());
}
```

The constructTower method has the parameters: marker position, marker
orientation and its PatternId. In the method the objects for the respective
tower type are created.

```cpp
if (markerId == 20){
    mTowerNumber++;
    mTower = VectorTower(towerPosition, towerOrientation,
        m_pSceneMgr, mTowerNumber);
    mTowers.push_back(mTower);
}
```

When the tower is constructed it is checked if the construction process
can continue or needs to be reversed. First the correct marker and tower
links are identified with the methods compareMarker2Tower and isWith-
InThreshold. The first method checks if the detected marker’s PatternId
equals the tower type. The second method checks if the marker is within
a certain tolerance zone of the tower object. This assures the tower’s con-
struction process can continue even when the marker is slightly moved.

```cpp
bool GameState::isWithinThreshold(CamVid::TrackCoord marker, RenderableObject object){
    if (marker.getMarkerPosition().x > object.getPosition().x - th){
```
if (marker.getMarkerPosition().x < object.getPosition().x + th){
    if (marker.getMarkerPosition().y > object.getPosition().y - th){
        if (marker.getMarkerPosition().y < object.getPosition().y + th){
            return true;
        }
    }
}
return false;

During the construction or deconstruction process the animation states and the ClockCycle budget are updated. In class Tower the towerStatus variable is changed between Compile, Decompile and Active and a variable compileTime keeps track of the construction process. The ClockCycles resource is likewise added to the tower’s own budget or refunded to the player’s resource pool, when the tower is deconstructed.

if (mTowers[i].hasMarker() && mTowers[i].isInFlux() == true && mCurrentClockCycles > 0){
    mTowers[i].setTowerStatus("Compile");
    mTowers[i].addActualCompileTime(mTimeSinceLastFrame);
    mTowers[i].addClockCycles(mTowers[i].getCost() / mTowers[i].getTotalCompileTime() * mTimeSinceLastFrame);
}

When the tower reaches the Active state it begins searching for virus targets using the check4TargetsInRange method and fires at it, if a target was found and the fire delay of the tower has passed.

for (int i = 0; i < mTowers.size(); i++){
    if (mTowers[i].getTowerStatus() == "Active"){
        if (mTimeSinceCpuBuilt - mTowers[i].getTimeOfLastShot() >= mTowers[i].getDelay()){
            mTowers[i].check4TargetsInRange(&mViruses);
            if (mTowers[i].hasTarget()){
                fireTower(&mTowers[i]);
            }
        }
    }
}

At the end of the handleTowers method the towers draw method is called.

3.5.3 handleCpu()

The central game structure, the player needs to protect from the advancing enemies is called cpu and is managed in its own method. Like the towers the cpu is created, by placing a marker with the appropriate pattern-id
on the playing field. The cpu’s pattern-id is fixed to 0, which is used for the world marker in the calculation algorithm of ARToolkit. The positions and orientations of the other markers are calculated based on the world marker’s own position and orientation.

If an enemy manages to reach the central structure it damages it, reducing the cpu’s remaining health. When this happens the enemy triggers the cpu’s hit method.

```cpp
void Cpu::hit(Virus* virus, float time){
    if(virus->isAlive()){
        mHealth -= virus->getDamage();
        if (!mBeingHit){
            mTimeOfHit = time;
            mBeingHit = true;
        }
    }
    if(mHealth <= 0){
        mIsAlive = false;
    }
}
```

To visualize the damage two graphical changes in the cpu object take place. At the actual moment of the hit the cpu’s texture is changed, by setting a new material, for a short time to achieve a flash-like effect. The second visualisation represents the overall status of the cpu. For every 20% of its health lost the cpu changes color to give the player an indication of the health level. These operation all use the setMaterial method in the Cpu class, where the material changes are made to the two entities that constitute the cpu.

```cpp
void Cpu::setMaterial(string materialName){
    Ogre::MaterialPtr materialPtr = Ogre::MaterialManager::getSingleton().getByName(materialName);
    mActiveMaterialName = materialName;
    mEntity->getSubEntity(0)->setMaterial(materialPtr);
    pyramidMaterialName = materialName;
    pyramidMaterialName.append("Pyramid");
    materialPtr = Ogre::MaterialManager::getSingleton().getByName(pyramidMaterialName);
    getByName(pyramidMaterialName);
    mEntity->getSubEntity(1)->setMaterial(materialPtr);
}
```

The time passed after a hit and other time based calculations in the GameState are based on the time passed since the cpu went active/was built, which is saved in mTimeSinceCpuActive/mTimeSinceCpuBuilt during the update() method.

Another task of the handleCpu method is the creation and management of two more game objects, the clock cycle display and the lanes. The former is a part of the cpu that is responsible for visualizing the amount of clock
cycle resource the player has left, represented by a column of blocks on every side of the cpu that lights up/darkens when the resource level changes. This is again achieved by changing the material for the respective blocks. This functionality is managed by the handleCCDisplay method. The lanes visualize the path the enemies take on their way to the cpu. They are constructed with the method constructLanes, when the cpu changes to active status.

3.5.4 handleViruses()

The enemies in the tower defense game are called Viruses and it is the player’s goal to prevent them from reaching the central structure as long as possible. The abstract class Virus declares the necessary variables for all viruses, such as the damage they deal, the speed at which they move and the direction they are headed for. Methods include generateName, which generates a unique virus name to identify the object and creates named SceneNode and Entity objects, and the hit method that is called when a virus reaches the cpu. Specific viruses are child classes of Virus. The number and formation in which the viruses approach are not fixed, as the game is supposed to be easily expandable allowing additional virus types to be added later. Instead the virus formats are entered prior to game launch in four txt files, one for each direction (named North.txt, East.txt, West.txt, South.txt). There the Virus type and number can be entered. The format for the entries is: The time at which, the Virus line should enter the game (in seconds passed since the cpu went active) followed by the actual viruses(identified by their name), which will be created in game at the specified time. An example for a virus configuration file, e.g. North.txt:

5
Grunt
9
Grunt, Grunt
13
Grunt, Grunt, Grunt

When the game is started and the enter() method of the GameState object is called, the virus files are read. During this process Line objects are created from the virus entries containing virus type, number, the time of their planned creation and their position in the formation. The latter is calculated based on the virus’ width, which is defined in their respective class. The resulting Line objects are saved in a queue, which is then evaluated in handleViruses’ processLine method. There the front line virus is spawned at the appropriate time and the corresponding line object is deleted from the queue.

```cpp
void GameState::processLines(std::queue<Line>* lines)
{
    if(!lines->empty()){
        if (mTimeSinceCpuActive*0.001 >= lines->front().getTime() ){
```
spawnViruses(lines->front());
lines->pop();
}
}
}

In spawnViruses the Ogre SceneNode and Entity objects for the viruses are created and their final position relative to the cpu are calculated. The virus instances are stored in a map container, so they can be easily found and deleted in handleProjectiles().

Once they are in-game, the viruses are moved when their draw method is called, which is located in MoveableObject for all Viruses (but could be individulized in the specific virus class if the virus is supposed to have a unique way of movement). There the translateVector used to move them per frame is calculated with the time passed since the last frame and the virus’ scene node itself is translated.

void MoveableObject::draw(float timeSinceLastFrame){
    mAnimationState->addTime(timeSinceLastFrame*0.001);
    if(mIsAlive){
        mTranslateVector = mDirection * mSpeed * 
            timeSinceLastFrame*0.001;
        mSceneMgr->getSceneNode(mName)->translate(mTranslateVector);
    }
}

To check if a virus has reached the cpu the check4Collision method is used. It uses Ogre’s AxisAlignedBoundingBox class to create a bounding box query around the virus’ scene node that returns the entities contained in the given bounding box.

void GameState::check4Collision(){
    Vector3 cpuUpVector = m_pSceneMgr->getSceneNode("Cpu")-
        getOrientation() * Vector3::UNIT_Y;
    Vector3 cpuPosition = m_pSceneMgr->getSceneNode("Cpu")
        ->getPosition();
    Vector3 AAVector1 = cpuPosition + cpuUpVector.normalise() * 15;
    Vector3 AAVector2 = cpuPosition - cpuUpVector.normalise() * 15;

    AxisAlignedBox box = AxisAlignedBox( AAVector2, AAVector1);
    AxisAlignedBoxSceneQuery* virusQuery = m_pSceneMgr
        ->createAABBQuery(box,VIRUS);
    SceneQueryResults& virusQueryResult = virusQuery->execute();
    SceneQueryResult::iterator it = virusQueryResult.movables.begin();
    for (; it != virusQueryResult.movables.end(); it++){
        mCpu.hit(&mViruses.find((*it)->getName())
            ->second, mTimeSinceCpuBuilt);
        mViruses.find((*it)->getName())->second.setIsAlive(false);
    }
To test only for the cpu entity, the query receives a mask parameter, which specifies the objects that are returned by the query. The masks that can be used in the query test are defined by an enumeration:

```cpp
enum QueryFlags{
    TOWER = 1<<0,
    VIRUS = 1<<1,
    CPU = 1<<2,
    LANE = 1<<3,
    PROJECTILE = 1<<4
};
```

If a collision with the cpu entity is detected the cpu’s hit method is called with a pointer to the triggering virus and the time of the collision, which is used in the Cpu class for the calculation of the material change duration. When a collision with the cpu occurs or the virus is destroyed by a projectile the mIsAlive bool is changed and the timeOfDestruction variable is used to measure the duration of the destruction animation. When the animation has finished the virus object is destroyed.

```cpp
if (!(*mVirusIterator).second.isAlive()){
    if ((*mVirusIterator).second.getAnimationState()->
        getAnimationName()!="Destruct"){(*mVirusIterator).second
        .setTimeOfDestruction(mTimeSinceCpuBuilt);
        (*mVirusIterator).second.setAnimationState("Destruct");
    }
    if ((*mVirusIterator).second.getAnimationState()->
        getAnimationName()=="Destruct" && mTimeSinceCpuBuilt -
        (*mVirusIterator).second.getTimeOfDestruction() >=
        (*mVirusIterator).second.getTime2Destruct()){  
        mVirusIterator->second.unregister();
        mViruses.erase(mVirusIterator++);
    }else{
        ++mVirusIterator;
    }
}else{
    ++mVirusIterator;
}
```

### 3.5.5 handleProjectiles()

Objects of the Projectile class are created when a tower has acquired a target and fires at the virus using its fire method. Similar to Virus, the parent class for Projectile is MoveableObject. The instances of Projectile are stored in a vector called mProjectiles located in GameState.

```cpp
30
```
Like the viruses the Projectile class uses a bounding box query to detect collision. In this case the mask used for the test is VIRUS, and a bounding sphere is used for the test. If a virus is detected, the targeted virus’ hit method is called. A Projectiles stores a pointer to the target virus and its tower of origin and is destroyed alongside its target, should the latter be destroyed by another projectile or by reaching the cpu.

```cpp
void Projectile::check4Collision()
{
    Sphere sphere = Sphere(mSceneMgr->getSceneNode(mName)->
        getPosition(), Ogre::Real(0.0001));
    SphereSceneQuery* projectileQuery = mSceneMgr->
        createSphereQuery(sphere,VIRUS);
    SceneQueryResult& projectileQueryResult = projectileQuery
        ->execute();
    SceneQueryResultMovableList::iterator it =
        projectileQueryResult.movables.begin();
    if (!projectileQueryResult.movables.empty()){
        for (; it != projectileQueryResult.movables.end(); it++){
            if ( (*it)->getName() == mTargetName ){
                mTarget->hit(mOriginTower);
                mIsAlive = false;
            }
        }
    }
}
```

Another method that is called every frame in handleProjectiles is updateDirection. It is used to adjust the direction of the projectiles to the new position of the target. GetDerivedPosition() is used here to retrieve the position of the virus object in world coordinates, instead of the virus’s local coordinate system.

```cpp
void Projectile::updateDirection()
{
    mDirection = mSceneMgr->getSceneNode(mTarget->getName())
        ->getDerivedPosition() - mSceneMgr->getSceneNode(mName)
        ->_getDerivedPosition();
    mDirection.normalise();
}
```

A projectile is destroyed when its Virus target is destroyed or when it collides with its target. When this happens the projectiles Destruct animation is played for the amount of time specified in the class, similar to the Virus class. After the time has passed the projectile object is destroyed.

Additional projectile types can be added by inheriting from the projectile class and can incorporate their own methods e.g. for collision detection to achieve different projectile effects. For example a bigger rectangle shaped bounding box could be used to target multiple virus target at once etc.
4 Results

The goal of this work was to create a game that would allow users to experience Augmented Reality interaction in a playful, direct way, while offering gameplay that is suited to for the Augment Reality environment. The result is a game that combines the Ogre 3d engine with the marker-tracking of ARToolkitPlus and features gameplay that is designed around marker interaction. All planned features were integrated in the allotted time and the game can be easily expanded by adding or modifying the appropriate classes and supplying additional assets (new meshes and animations for additional tower and virus types).

4.1 Installation

For its marker tracking the game requires a camera connected to the system. The game was tested using a Logitech QuickCam Pro 9000 USB 2 camera. Other webcam types could not be tested, but every camera that is suitable for tracking with ARToolkit and ARToolkitPlus should achieve good results. For camera configuration of the ARToolkitPlus a configuration file is required, which is located in the game directory. The Logitech QuickCam Pro 9000 is used as default (camera_para.dat) and two additional configuration files are available: A file for the Sony EyeToy camera (camera_para.dat_EYETOY) as well as the standard camera calibration file for ARToolkitPlus (camera_para.dat_STANDARD), which should achieve good results for most webcams.

Marker tracking also depends on the size of the markers used. The default size for the markers is 80 mm and the marker files can be found in the ARToolkitPlus distribution. The standard ARToolkit bch markers with thin border are used by default. Marker-ids used are 0 (for the cpu) and 20 (for towers of the VectorTower type). If there are multiple webcams present in a system, as might be the case with built-in webcams in notebooks, the webcam that will be used in the application can be specified in the arbackground.material file in the media sub-directory.

4.2 Gameplay

The default setup for which the game was designed has the player sitting at a plain surface with enough room for several markers on it, with a display in front of him. The camera should be placed on top of the display or another elevated position to have a clear view at the playing-field. After entering the game through the menu screen, the game is started by placing the cpu marker with pattern-id 0 on the playing field. After the cpu is constructed the virus timers are started and the enemy waves will start to appear, depending on their settings in the respective configuration files.
(North, West, South and East .txt files in the game directory). To prevent the viruses from reaching the cpu the player constructs towers by placing markers with the pattern-id 20 on the playing field. This starts the construction process of the tower, which will fire at the viruses once it is completed. The maximum number of towers the player can construct is limited by the available clock-cycles resource, which is visualized by the yellow columns in the middle of the cpu structure. The remaining health is visualized by the color of the cpu structure, which changes from blue to red the more damaged the cpu becomes.

The game ends if the cpu is destroyed. The game continues to run even if all virus waves specified in the configuration files have been spawned to allow free interaction with the scene.

Figure 16: screenshot of the game
5 Conclusion

As expected the integration of all the different components used for this work proved to be the biggest challenge. Familiarization with ARToolkit-Plus’ code along with the integration into Ogre made up a big part of the time. Martina Brümmer’s work offered a good documentation to a similar project, which was very helpful. For the Ogre integration the community forums held some very informative threads, most of all posts of the user Futnuh who documented his earlier experiments with ARToolkitPlus and Ogre.

The Ogre Wiki and the book Pro Ogre 3D Programming by Gregory Junker [12] provided documentation for the Ogre engine and a former course visit for Maya animation helped with creating the necessary meshes and animations.

Overall the work provides a good basis for the use of ARToolkitPlus with the Ogre engine, while allowing interesting gameplay and interaction in an Augmented Reality environment.

5.1 Prospect

While all the project’s planned features were implemented and the game is functional as is, there are many additional features that would improve upon the basis created in this work.

First of all additional game elements would help diversify the gameplay and allow for more varied ways to play the game. More virus types and tower varieties could easily be added to the game’s existing structure. Ideally their addition would lead to new and interesting interaction for the player. Examples for this could be:

- Additional virus types with different attributes. Slow and tough viruses that require several hits etc.

- An enemy that emits a field around itself in regular intervals (about 5-10 second) that stuns all active towers in its radius and prevents them from firing for a few seconds. To counter this, the player could occlude the threatened towers for a brief moment to allow the field to pass without stunning the towers.

- Several tower types could be added that target and fire at enemies in a different way, damaging multiple viruses at once, slowing them etc.

- A reconnaissance tower that does not fire but provides a lane warning effect, that flashes a lane’s color when viruses are approaching in the next few seconds, changing color the more viruses are approaching. This would allow the player to sacrifice clock cycle resource and building space for additional information.
Another game mode could be implemented to allow for some kind of tutorial, which would help if the virus and tower number increases. There the player would be allowed to inspect viruses moving around the screen by catching them with a marker, receiving additional information about this enemy type. On the code side this addition could be implemented by changing the handle methods into manager classes and using their functionality for a new GameState class.

Other elements that would add to the game are a high score system and a detailed game over screen, which give information about the player’s progress. Sound effects and music were omitted in this work due to time constraints but could easily be added to the game’s structure.

Another aspect that could not be realized in the allotted time are user tests that could help fine tuning the interaction process. The difference in orientation from the camera image and the look at the markers in front of the players, might lead to confusion; a mirror mode option that switches the handedness of the scene could alleviate this.
References

[1] Sony Computer Entertainment, The Eye of Judgment Homepage
http://www.eyeofjudgment.com/ [Online February 2010]

[2] Julian Oliver, levelHead
http://ljudmila.org/~julian/levelhead/ [Online February 2010]


http://www.wikitude.org/ [Online February 2010]

[5] ARToolkit documentation
http://www.hitl.washington.edu/artoolkit/documentation/
[Online February 2010]


[7] Ogre Community, Ogre Webcam Plug-in
[Online February 2010]

[8] Ogre Community, Advanced Ogre Framework
[Online February 2010]

[9] Ogre Community, ARToolkitPlus
http://www.ogre3d.org/forums/viewtopic.php?f=5&t=22584&start=0
[Online February 2010]

[10] Paul Preece, Desktop Tower Defense

[11] PopCap Games, Plants vs Zombies