Final Report

The Cave of Caerbannog

By
Team MIGI2
Taico Aerts, taerts, 4345797
Chiel Bruin, cbruin, 4368436
Bram Crielaard, bcrielaard, 4371755
Wytze Elhorst, welhorst, 4306228
Robin van der Wal, robinvanderwal, 4142918

Context project 2015-2016

FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND COMPUTER SCIENCE.

Instructors: prof. dr. A. Hanjalic and dr. A. Bacchelli

Context coordinator: dr. ir. R. Bidarra.

Teaching assistants: S. van den Oever and J. van Schagen

June 2016

Table of contents

1 Introduction

This document is the final report of the award winning game “The Cave Of Caerbannog”, created by *MIGI2* for the TU Delft context project Computer Games. For this project we were assigned the task of creating an “asymmetrical multiplayer virtual reality game for the Oculus Rift”. We were given the following requirements to make this happen:

- *Java*
  The game must be written in Java. The static analysis tools Maven, CheckStyle, PMD, and FindBugs have to be used to guarantee a high quality code base.
  The only way we could make this happen would be to write the game in Java, which is exactly what we did. As all members of the development team had worked with java before this was the most suitable language, as we would not have to learn a new one.
- *Oculus Rift*
  One player of the game should be immersed in a virtual world and should have a special role in the game. Therefore, a game must be developed where one person is playing in a virtual world, while other people are able to interact with that world in a different, non virtual reality, way.
  An Oculus Rift was provided to test the game on. After extensive research, we managed to find the correct combination of hardware, drivers and software to be able to run our game with the Oculus.
- *Multiplayer*
  There should be one or more people playing the game with (or against) the player wearing the Oculus Rift. Other than the Oculus itself, *low-budget* technology has to be used to make this happen, as the game should not become more expensive because of extra peripherals needed.
  As most people own a smartphone with HTML5 support we decided to create a web interface for the game, which can be accessed from a phone. This was extensively tested on devices ranging from iPhones to BlackBerrys and worked perfectly on most of them. The only other peripheral we decided on was an Xbox360 controller to be used by the person using the Oculus Rift. As the Xbox360 controller is generally regarded as the best controller on the market to date plenty of people already own one.
- *jMonkeyEngine*
  The engine the game should run on must be jMonkeyEngine. The version of the jMonkeyEngine that must be used allows developers to interface with the Oculus Rift using Java, and as a result is a suitable engine for the requested game.
  As we were required to write the game in java and use the Oculus Rift the jMonkeyEngine was decided to be used for the game. As a result we spent time learning how to use the jMonkeyEngine, which (as requested) was used as the engine for our game.

- *Fun*
  The game should be fun to play for everyone involved. This means the people playing from their mobile phones should not just be simple pawns in the game played by the person wearing the Oculus Rift.
  To make sure our game was fun we designed it so that everyone was needed while playing the game. After multiple playtest sessions (which we will expand upon in chapter 5) we ended up with a good balance for both our teams, which meant the game was fun to play.

In this document will go into more detail explaining how the game came to be. First, an overview of the developed product is given. Secondly, a reflection on the product from a software engineering perspective is made. Thirdly, a detailed functional description of the most intricate parts of the product is given. Then we explain our decisions from an interaction design standpoint followed by a brief evaluation of the product. And finally, we give an outlook on what can be added in the future.

2 Overview

This section will give an overview of the product that we created.

2.1 About the game

The Cave of Caerbannog is an asynchronous multiplayer game that uses virtual reality. It is set in a maze-like cave system, with a treasure at the end. There are three different roles. First, there is one player wearing the Oculus Rift, the Knight of Caerbannog. For the knight, the goal is to get the treasure at the end of the cave. Then there are two teams that use their mobile devices to connect to the game; the elves, who will assist the knight, and the dwarfs, who will work against the knight. The elves win if the knight succeeds, while the dwarfs win if the knight fails to get to the treasure before the cave collapses or if the knight perishes along the way.

2.2 Structure of the game

As a requirement of the Computer Games context, we have to use a game engine called jMonkeyEngine. The engine manages the interaction with the Oculus Rift, and thus the actual rendering of our virtual environment.

Our game consists of two major components. First there is the part of the game that is responsible for creating the environment, the game logic, and the interaction with the knight. This part interfaces directly with the engine. Then there is the web interface, which consists of an embedded webserver and a client side implementation.

The webserver manages the interaction between the game and the elves and dwarfs. Every client that connects to the webserver is attached as an observer to the game thread.
When a client performs an action, a message is sent from the client to the webserver. The webserver then decides if this action is valid, and if so, performs the action by interfacing with the game.

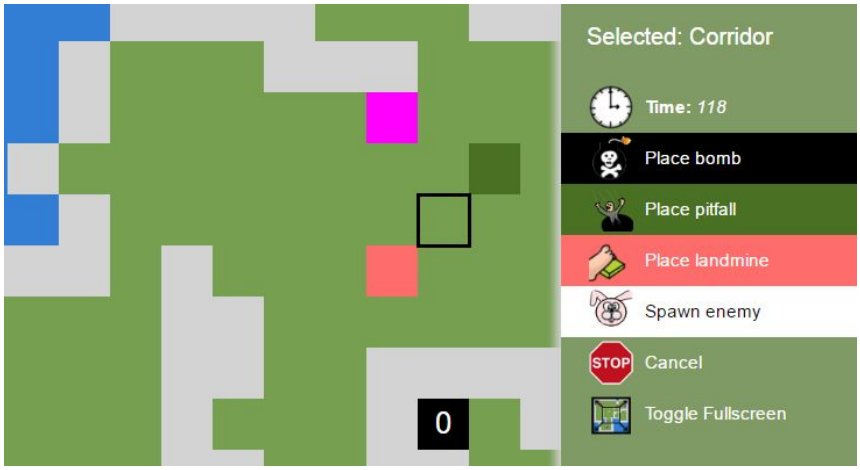

Figure 1: *The web interface for players in the dwarfs team. On the left you can see the map, with a bomb (black), pitfall (dark green), landmine (red/pink) and the player (pink/purple). On the right is the menu with the time remaining and the different actions that can be performed on the selected tile (outlined in black).*

2.3 Gameplay

The gameplay focuses on communication between the knight and the elves, and on competition with the dwarfs. The knight has to get through the maze to reach the treasure. The elves can see certain things that the knight cannot see or hardly see. For example, the elves can see pitfalls, which are invisible to the knight. There are also dangers and blockades that the elves will need to clear for the knight. For example, gates need to be opened by elves. Since the dwarfs will be constantly placing traps throughout the game, the elves and the knight will need to work together and communicate properly if they want to win. This creates fun and interesting gameplay, that is well suited for quick matches as played on parties.

3 Reflection

This chapter will give a reflection of the project from a software engineering perspective.

3.1 The Product

Overall, we are happy with the resulting software product. We have used various different design patterns, we have we have a test coverage of about 90%, and we believe that the code is of high quality.

Design patterns used include Singleton, Abstract factory, Observer and Strategy. Singleton is used for the Main class and the audio engine components. Abstract factory is used for level creation. The observer pattern is applied over large parts of the code, and the strategy pattern is used for the AI/Control of moving entities.

Of course, there are always things that can be improved. Currently, we have a high coupling with our Main class. Any class that needs to interact with the game itself has to acquire an instance of the game via Main.getInstance().getCurrentGame(). Any entity that needs to interact with the player needs to obtain this reference via this instance of the main game.

3.2 The Process

The process of designing and creating our game went very well. We gradually expanded the codebase, while maintaining a high test coverage. Our strict pull request review policy ensured that all code was checked by at least two members, which meant that a lot of mistakes were found and fixed before they were merged.

During the entire project, we have made various large refactorings to improve the code quality and to reduce the entropy in our code. For example, the framework was refactored in sprint 3, to improve the package structure.The layout of our javadocs and comments were generalized in sprint 5. The names of all variables in the code were refactored in sprint 7, so that no variables are ambiguous, and various refactorings occurred in sprint 8, to improve package structure, fix layout mistakes and to normalize code style.

We also tried our best to improve our code based on the weekly feedback of our Software Engineering TA and the output of the static analysis tools, Checkstyle, PMD, Findbugs and SIG.

The main lesson that we learned, is that web development can be tricky, and requires a lot of time. The main issue is to support different browsers, while maintaining short and somewhat readable code. As different browsers take different approaches on how to render certain elements, some “tricks” need to be applied to make it work in all of them. When you create a website, you usually use a framework like bootstrap or modernizr. However, when you create an interface for a game, you need specific elements that are not normally used in websites. This means that you have to create them mostly from scratch.

4 Functionality Description

Our game leans heavily on two parts of the code that are responsible for the two core mechanics in the game. These parts are the maze generation and the webserver. The maze generation ensures the replayablility of the game and also provides interesting challenges for the players. The webserver is responsible for the multiplayer part of the game by providing access to the game for the mobile users. The following sections will quickly describe these parts of our program. A more elaborate explanation of them and some other smaller sections, can be found in the architecture design document.

4.1 Maze generation

The random maze generation has the task to generate a new maze for every run of the game. Creating a new maze ensures that the game is a different experience every time you

play it and therefore that it stays fun, even when it is played extensively. A big challenge in generating the mazes is to ensure that the player can always get to the treasure. We synthesized a few rules that the maze generation has to adhere to to ensure this:

- There is always at least one path from the start to the finish.
- A level must consist of rooms connected by corridors.
- A room should be loaded from a file.

This last rule makes it possible to create multiple (preferably as much as possible) rooms in advance that are hand crafted. By creating those rooms by hand, we can create interesting puzzles and challenges that wouldn't be possible to generate using a room generating algorithm and therefore make the game more interesting.

The algorithm for creating the maze is in short a deliberately broken minimum spanning tree (MST) algorithm. In a normal MST-generating algorithm, exactly one path is possible from each node to any other node. The algorithm we used, however, leaves some extra edges in the graph to allow for multiple correct paths from the starting room to the treasure. Then each node must be converted to a room in the maze. The last step is converting the edges between nodes to corridors, connecting different rooms. This is done by first connecting them with corridors using breadth first search. To give the corridors a cave-like feeling, the paths are widened using an exponential distribution for each direction. This results in more jagged walls, which gives the maze the cave like feel.

4.2 Webserver

A big part of our game relies on the interaction between the main player inside the cave and the other players on their mobile devices. To make this interaction possible for all types of devices (Android, iOS, etc.) we decided to use a web application that is platform independent. Another advantage of this method is that our game can be played without installing a native application on the device. Providing the web app to the users raised the need to run a webserver that could serve the web pages to the users and allow them to perform actions on the virtual world. This server is embedded in our game and automatically starts with it. When the mobile users connect to it, the map of the maze is shown. Dwarfs will see the entire map, but elves will only see the explored parts of the cave as they do not live in the cave and have to explore the maze together with the knight. On this map all game elements are shown (except the treasure, as this will remove the exploration element of the game) and updated when they appear, move or disappear from the cave. The mobile users can tap on the map to perform various actions that have some interaction on the cave. For example, an elf could drop a carrot to distract the killer bunnies from the knight.

As all actions have cooldowns and need to be placed at least a certain distance from the player, the server has to perform various checks on each action request form a player. Without these checks a mobile player has unlimited opportunities to help / obstruct the knight, removing the fun of the game and making the knight extremely weak as he cannot move without triggering traps.

5 Interaction Design

The most important part of developing a game is making sure that it is fun to play and that people want to play it. The best way to make sure that people like your game is to let them play your game and tell you what they think about it, that's why during the final four weeks we organized playtest sessions during which five people got to play our game and give us feedback on what they thought about the game. We had them fill out a feedback questionnaire so that we could store the feedback they gave us. We also made notes about every gameplay related comment that people made, so that we could discuss them afterwards,

5.1 Playtest Results

Our first play session had a few bumps on the road, as the game crashed a few times during the session, however the people playing did get to play long enough to get a good impression and to give us adequate feedback. The most prominent part of the feedback was that the game was very unbalanced since the dwarfs had too many strong tools, also the elves didn't have a lot to do during the game.

A week later during our second playtesting session things went a lot smoother and people got to play our game for a good amount. The feedback we got this time told us that the bunnies were too strong and we noticed that a lot of people didn't clearly know what they should be doing at the start. For a graph containing our playtest results, please refer to **Appendix A**.

5.2 Solutions

After the first play session it became clear that the dwarfs were a lot too strong. The solution to this was quite simple as we gave the actions of the dwarfs a larger cooldown on their actions and added a radius around the player in which the dwarfs are unable to place traps. The second play session taught us that we needed to make the explanation of the game clearer and make the bunnies weaker. We solved the former by creating a tutorial level for the oculus rift player and a written tutorial for the web interface. We limited the power of the bunny by raising the cooldown of the bunny..

5.3 Nausea reduction

One of the major drawbacks of the oculus rift, is that it can be very nausea inducing. This is a serious issue for game development, because if players get nauseous while playing your game, they won't want to play your game anymore. We have accepted the fact that we can't remove nausea induction in its entirety, as this is a limitation of the platform, but we tried to prevent it as much as possible.

We found a few things we could implement in our game to reduce nausea. The first thing we found was that the slower the player walks through virtual reality, the less likely they will suffer from nausea (Riftinfo, 2015). This is especially true when turning your camera

around (Will Mason, 2015). So the obvious solution was to reduce the movement speed of the player and to severely reduce the turning speed of the camera.

Another method of nausea reduction is to add static objects in front of the camera where the player can focus on (David Whittinghill, 2015). An example of this is having a nose on the screen which you can focus on, we added a nose to the HUD of the player. To keep in theme with the player being a knight, we also added a helmet. These objects don't decrease the vision of the player too much, and after some testing we did discover that they do help a lot with the reduction of nausea.

5.4 Product Design

We decided against having a real menu in our game, as this breaks the immersion. Instead when you start the application, you will immediately be put in the game in a "menu level". This has the advantage that it will get the player familiarised with the core game aspects, such as the controls and the environment.

The player will also notice there are two big banners with special tiles in front of them. The first banner says "tutorial" and when a player walks onto the tile in front of it, the player will be put into the tutorial level. The tutorial has the same philosophy as our menu level, the player is free to do what they want in the this level, but in each room there are posters explaining what a certain object does, as well as this object which the player can interact with. After finishing the tutorial, the actual game starts.

The second banner is locked behind a door and says "start". This door can be opened by picking up the key found in the room and when the player walks on the platform, the actual game will start immediately. This means that players who know how the game works can quickly open the door and start playing the game without having to go through the tutorial every time, but new players that don't know anything about the game, are forced to go through the tutorial since they don't know how to open the door.

The players on our web interface will see two buttons after loading the interface. One of these buttons will take the player to the tutorial for the web interface, which will briefly explain which actions they will be able to perform. The other button will let them join the game, at which point they will be able to join a team. Once they have joined the team they will see the player walking around in the menu level. Here they have the chance to test out all their actions on the player, so that they will get a feel of what they can do. Once the player gets into the tutorial these actions are disabled until the game starts.

The main goal of this design is to let the players experiment what they can do on their own, but also offer them the opportunity to learn how to play during the interactive tutorial. This will keep the attention of the players and if they get bored of the tutorial, they can just walk straight through to start the actual game.

6 Evaluation

In this section the two main evaluation methods are briefly described with their results. First the gameplay testing is discussed, followed by the failure analysis of the product.

6.1 Gameplay testing

We wanted to create the most fun experience that we were able to deliver, however we cannot check this ourselves. As said earlier this was done by holding play test sessions with people who had never played the game before. Play testing allowed us to evaluate newly implemented or improved features in our game. There was a big difference between what we, the developers, expected to go down and what actually happened. This was mainly because we as developers know how the game works and how it is meant to be played, but people who have never seen the software before have no idea. This resulted in us spending a lot of extra time on creating a tutorial and trying to improve the clarity of the interface. Thus, gameplay testing helped a lot with improving the product.

6.2 Failure Analysis

The main tool that we used for failure analysis was unit testing. We were instructed to use line coverage of Cobertura to measure coverage and all groups were required to at least cover 75% of all produced code. However, we as a group wanted to go even higher and aimed at 85%. In the end we managed to hit around 90%, a number that we are quite proud of, as it not the easiest to test a game that mostly consists of a graphical user interface. Below is a figure of all packages and their line coverage.The hardest classes to test were the main class and some of the control classes. It also didn't make much sense to test these classes as they are largely dependent on other classes.

As a result, our game works as designed. As you cannot prove the absence of bugs, there is always the possibility for bugs to be in the game. However, none of them were found at the end of development.

Package	# Classes	Line Coverage	
All Packages	124	90%	3309/3649
nl.tudelft.contextproject	2	53%	91/170
nl.tudelft.contextproject.audio	3	90%	101/111
nl.tudelft.contextproject.controller	9	89%	185/207
nl.tudelft.contextproject.hud	1	90%	132/146
nl.tudelft.contextproject.input	1	88%	30/34
nl.tudelft.contextproject.logging	3	100%	79/79
nl.tudelft.contextproject.model	6	100%	65/65
nl.tudelft.contextproject.model.entities	10	86%	247/286
nl.tudelft.contextproject.model.entities.control	4	61%	55/90
nl.tudelft.contextproject.model.entities.environment	8	94%	224/236
nl.tudelft.contextproject.model.entities.exploding	3	94%	111/118
nl.tudelft.contextproject.model.entities.moving	3	93%	147/157
nl.tudelft.contextproject.model.entities.util	6	100%	54/54
nl.tudelft.contextproject.model.level	11	96%	417/432
nl.tudelft.contextproject.model.level.roomIO	5	94%	112/119
nl.tudelft.contextproject.model.level.util	19	97%	536/550
nl.tudelft.contextproject.util	7	85%	157/183
nl.tudelft.contextproject.webinterface	10	96%	392/406
nl.tudelft.contextproject.webinterface.util	4	94%	102/108
nl.tudelft.contextproject.webinterface.websockets	5	83%	68/81

Figure 3: the test coverage in all packages of "The Cave of Caerbannog", generated by Cobertura. The overall test coverage can be seen on the top line with "all packages."'

7 Outlook

In this chapter we will discuss what we would improve upon given more time, and how we would achieve these improvements.

7.1 Looks

One of the major things we would like to improve would be the look of our game, especially the look of the web interface. The web interface shows an incredibly simplified view of the game, resulting in people having to learn which colours depict which elements they represent in the game. From our play test sessions we have learned that people are satisfied with the way the interface looks, however a clearer and nicer looking interface would most likely improve the game.

The actual in game view could also be improved. All current models and textures are perfectly fine. However if you were to compare them to a AAA game, or even one of the more popular indie games, they fall flat. Improving these would immediately change the game from looking like a university project to looking like a proper, well developed game.

7.2 Gameplay

During the brainstorming phase of our game, we discussed numerous gameplay elements we thought would be interesting. For example, moving platforms, where the Oculus Rift player can stand on a platform which an elf can then move to a different location. However, multiple features, including the previously described one, were scrapped from the final product as it was unfeasible to include these mechanics in the given time. Given more time we would like to implement these features, as we all still agree they would make for a nice addition to the game.

7.3 Sounds

Currently our game has a minimal amount of sound effects. As we do support them we would like to create more custom sound effects, so every action has its own distinct sound. We would also like to have multiple background music tracks, as we currently only have one track which we loop indefinitely.

7.4 Rooms

Our maze is randomly generated from a fixed set of rooms. Having more time to create these rooms would increase the replayability of the game and as a result would improve the fun factor of the game.

7.5 Optimizations

Given more time we would look into ways to optimize parts of our game. For example: the jMonkeyEngine engine does not have a built in octree, which means all objects are rendered in their entirety, even if they are located an infinite distance away from the player. Being able to limit what is rendered by the engine would greatly improve performance, which means the game will run smoothly on less powerful computers.

Appendix A: Playtest Results

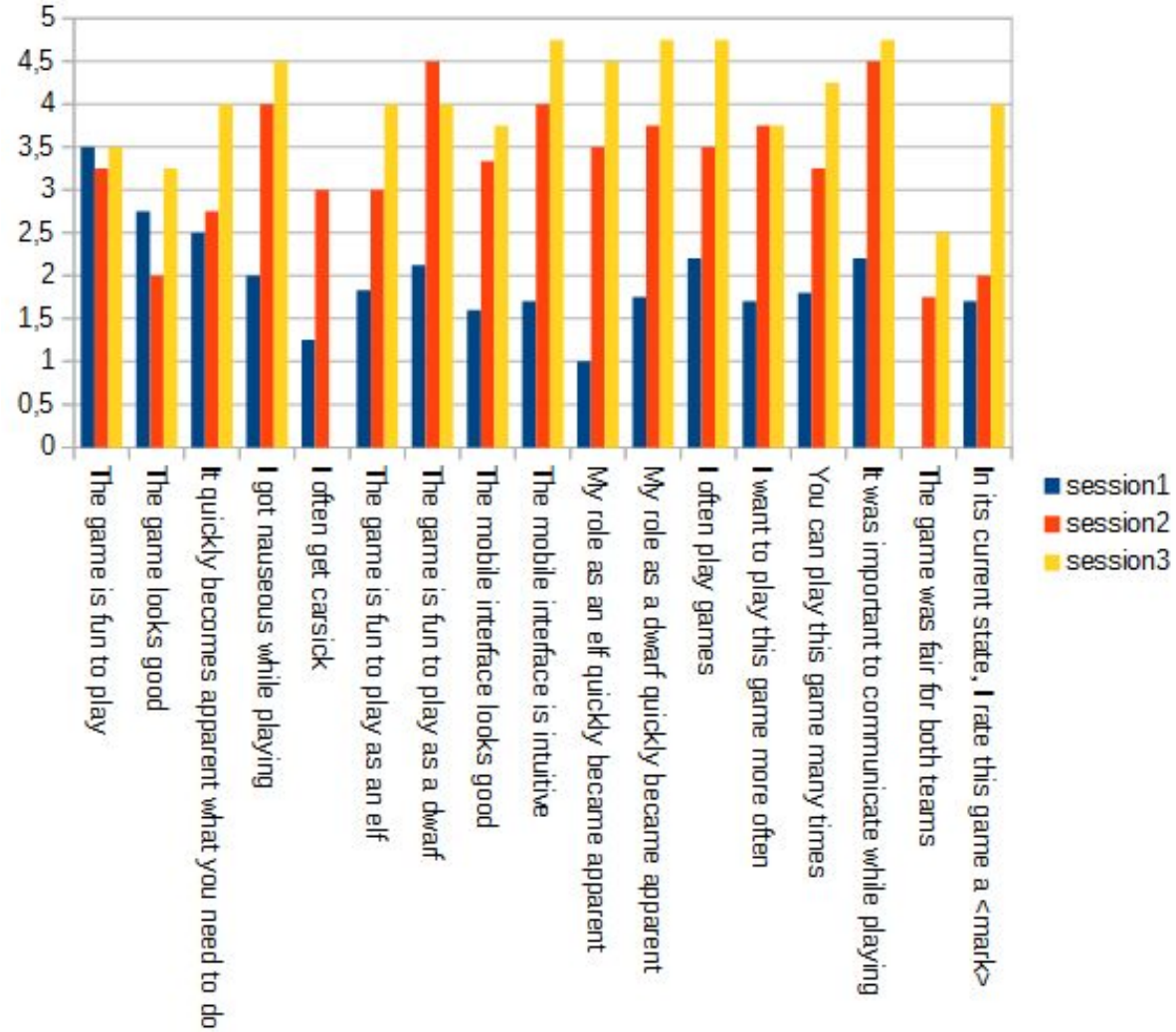

Figure A: The average grades given by our participants during our three recorded playtest sessions.

Appendix B: References

David Whittinghill,(2015). *Turns out the answer to virtual reality sickness is right in front of your face.* Retrieved from: http://theconversation.com/turns-out-the-answer-to-virtual-reality-sickness-is-right-in-front-of-your-face-41482

Riftinfo. (2015). In *Oculus Rift VR motion sickness - 11 ways to prevent it!* Retrieved from: http://riftinfo.com/oculus-rift-motion-sickness-11-techniques-to-prevent-it

Will Mason. (2015). *Five ways to reduce motion sickness in Vr.* Retrieved from: http://uploadvr.com/five-ways-to-reduce-motion-sickness-in-vr