

Architecture Design

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

May 2016

Table of contents

1 Introduction	2
1.1 Design goals	2
2 Software architecture views	3
2.1 Subsystem decomposition	3
2.2 Hardware/software mapping	4
2.3 Persistent data management	5
2.4 Concurrency	5
3 Glossary	6

1 Introduction

This document will go into detail explaining how “The Cave of Caerbannog” is put together, what one would need to be able to run it, and what the main design goals behind the game are.

1.1 Design goals

In this chapter the main design goals for the final release will be discussed.

Manageability

As the game can change during development, or even after release, one of the design goals is manageability. To make the code more manageable it will be split in independent parts, which will work together to make the game run. Interfaces shall be used to make this happen. All components of the game should be interchangeable with different components which implemented the same interface.

Quality

The code must be guaranteed to be of high quality. This can be guaranteed because of the following agreements between members of the development team:

- All code, excluding the GUI, has to be tested to reach a line coverage of at least 80%. If this task is impossible an explanation will have to be given as to why this is the case.
- Git and Github will be used for all code related to the project. This ensures we can always revert to an older version if the code breaks, and that there is a single repository where all the code can be found. All pull requests will have to be reviewed and accepted by at least two members of the development team before they can be merged.
- All methods and classes have to include javadoc explaining their functionality.
- The code may not contain warnings generated by checkstyle/FindBugs/PMD. If a developer has a good reason to keep one of the warnings an explanation has to be given.
- The program will be built by a continuous integration system after every push to the online branch. Pull requests that do not build will not be accepted. Builds will fail when the code does not compile.

Reliability

The game must remain playable and not crash under normal circumstances. This goal is reached by extensive play testing and the aforementioned guarantee that the code must be of high quality.

2 Software architecture views

In this chapter the architecture of the game will be discussed. This will be done by decomposing the project into its independent subsystems, explaining which hardware is used and how it is used, and how concurrency problems have been dealt with.

2.1 Subsystem decomposition

The system is divided up into different subsystems according to the MVC pattern. This adds to our design goal of “manageability”. One of the reasons the MVC pattern is used is because multiple views will be presented on the same game state. The main reason the MVC pattern was chosen was because the development team is comfortable with this pattern, as they have used it numerous times before.

Model

This subsystem consist of the jMonkey engine, as required by the stakeholders. If during development the development team realises the jMonkey engine is lacking in required features they will be added to the model subsystem.

View

This subsystem will be responsible for the GUI. Included will be all classes relevant to the rendering of scenes, showing of images, and menus.

Controller

This subsystem will be responsible for handling the user input and interacting with the different controllers.

The exact package structure for all of these subsystems will be added once they have been decided by the development team.

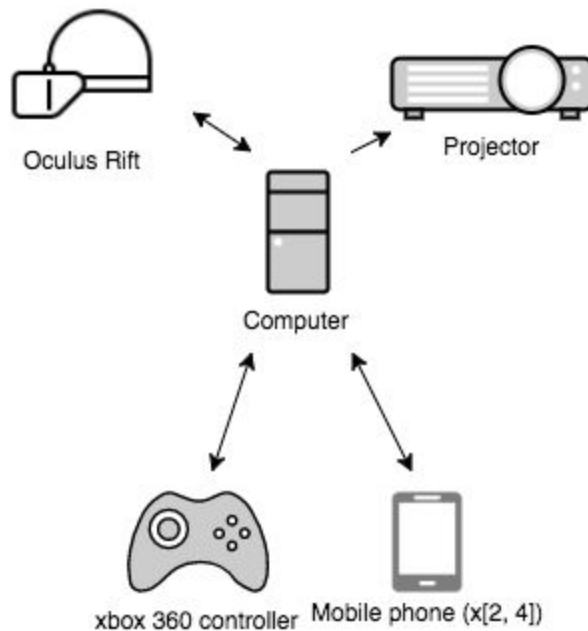
2.2 Hardware/software mapping

There are multiple hardware components which need to interact with each other. There is a single computer on which the game runs, which also acts as a server for the mobile phones. Connected to the computer are the following components:

- Oculus Rift (x1)
- Xbox 360 controller (x1)
- Mobile phone (x[2, 4])
- Projector (x1)

One player uses the Oculus Rift to view the game, and the xbox 360 controller to control the game.

Four other players use the projector to view an image of the map, and mobile phones to control the game. The mobile phones in turn display relevant information to what is currently going on and the relevant controls for the player.



2.2.1 Web Server

A large part of our game revolves around the multiplayer aspect. There will be one player that wears the oculus, and up to four players that participate in the game from their mobile phones. To communicate with these mobile phones, we use an embedded HTTP server.

The communication revolves around continuous status update requests from the clients to the server. The server will respond to these requests with relevant status information, with which the web page as seen by the clients is updated.

Communication

The communication between the server and the clients happens in the following way:

1. The webserver is started with the game
2. A client connects to server.
 - a. If the game is in progress, the client is denied from participating in the game with an error message.
 - b. If the game is full, that is, when four other clients have already connected, then the client is denied from participating in the game with an error message.
 - c. Otherwise, the client is added to the game, and is presented with a team selection page (2). The server will send a persistent cookie to the client for further tracking. This cookie will contain an unique id for this client.
3. The client selects a team
 - a. If the client does not have a valid unique id, then they are handled as in phase 1.
 - b. Otherwise, the server changes the team of the client to the selected team, and sends a confirmation.
 - c. The client starts a timer to send update requests on a set interval of 1 second.
4. When the client receives in a status update that the game has started
 - a. The client requests the map from the server.
 - b. The server sends the map encoded as a json object.
 - c. The client's web page is updated to show the map of the game, state (5).
5. When the client receives a status update while the game is in progress
 - a. The client's web page is updated to reflect the new changes.
6. When the client receives in a status update that the game has ended
 - a. The client's web page is updated to show the end game statistics
 - b. The client stops the timer, and no longer sends update requests to the server.

Status updates

Status updates are sent as json objects from the server. Status updates include the following information:

- The current game state (WAITING, STARTED, PAUSED, ENDED)
- The client's team (ELVES, DWARFS, NONE)
- If the client is in the ELVES team:
 - A list of all tile locations that have been explored.
 - A list of all entities on the map that an elf should be able to see.
- If the player is in the DWARFS team:
 - A list of all entities on the map that a dwarf should be able to see.

Client tracking

Since HTTP is a stateless protocol, and because ip addresses can change, we use cookies to track which request comes from which client. When a client connects to the game, and is allowed to join, the server sends a cookie with a unique id to the client. The client will in turn send this cookie to the server with each of their requests (this is done automatically by the client's browser).

Map

Below is an image that shows what a client “sees”, as it is at the end of sprint 3. Please keep in mind that this view will likely change when more of the functionality is implemented.

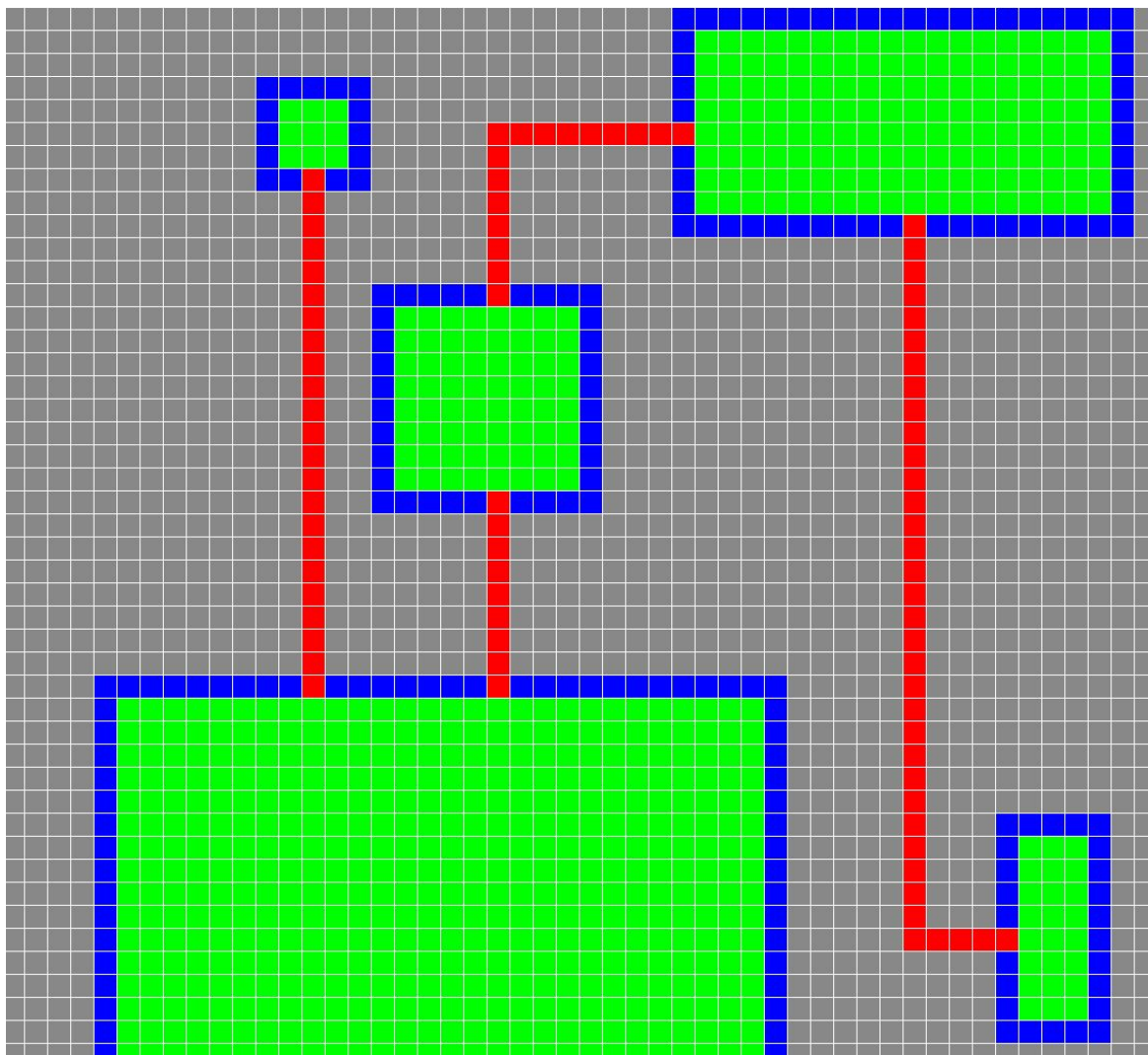


Figure 2: The map view of a web client.

2.3 Persistent data management

We will use persistent data management for two aspects of our game:

- The building blocks for the different levels, the rooms.
The rooms will be stored in xml/json files. These files will not change during gameplay, but changes can be made during development and playtesting.
- The game settings.
Settings are stored in a plain text format (yaml, xml, json or properties), and can be updated by the player in the menu.

We do not use any kind of Database Management System, since we do not have large amounts of data we have to store, and updates in the data are scarce.

2.4 Concurrency

As there are multiple people playing the game at the same time concurrency is a something we have to consider. As it is not possible to predict which concurrency problems will occur during the making of the game, this part will be expanded upon once the development team decides a way of tackling these problems.

3 Glossary

Branch

A split from the main code to which people can push changes and additions to the code.

Checkstyle

A static analysis tool which determines if the code complies with the style rules agreed upon by a development team.

FindBugs

A static analysis tool which detects possible bugs in java code.

Git

A version control system used for software development.

Github

A web-based git repository hosting.

GUI

Graphical User Interface, an interface which allows users to interact with electronic devices.

Merge

Blending of the code between two different branches.

MVC

Stands for “Model-view-controller”, A software design pattern for implementing user interfaces on computers. It splits the code into three different parts which all work together but could be interchanged for different parts.

PMD

A static analysis tool which uses a rule-set to determine if the code is erroneous.

Pull request

A request to add or change code and deliverables.

Push

Uploading code to a git repository.