# 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 |

June 2016

# Table of contents

# 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

### 2.1.1 Main framework

This section will explain the interaction between the controllers and the model, as this is the main framework of the game. For a more detailed overview of the structure of the program, see section 2.1.2 and see section 2.2.1 for the implementation of the web server.

**Game state**
Because the gameplay is divided in several separate parts we introduced GameStates. These states are:

- Waiting - The game waits for all players to connect and choose a team.
- Running - The main game is running
- Paused - The main game is suspended.
- Ended - The game is finished but not yet back in the waiting state for a new round.

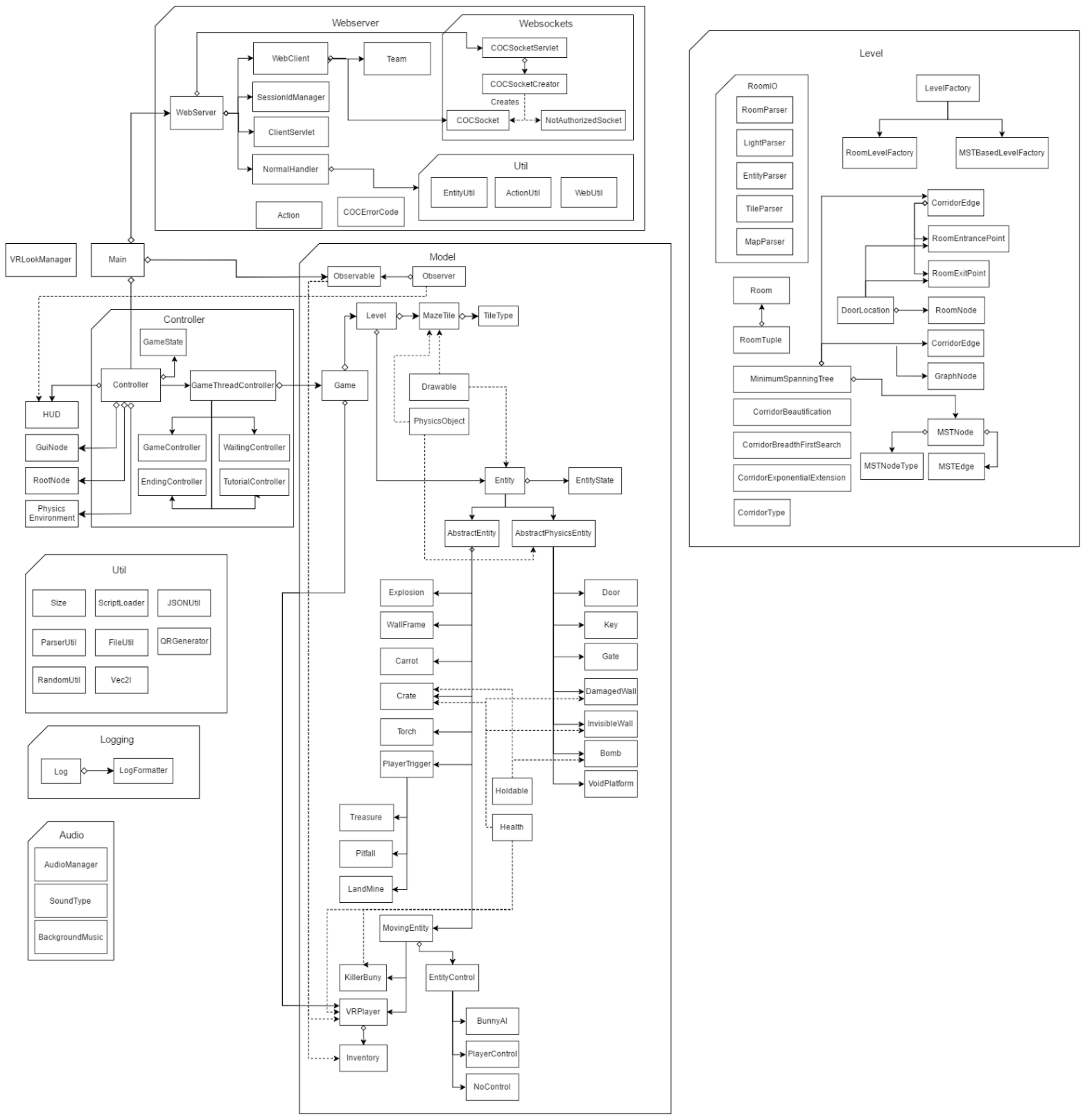
Each controller used in the program has a state that it manages. For example the WaitingController manages the waiting state. A controller also has the task to interface the renderer by providing methods to add drawables and Gui elements. An important part of adding and removing drawables is to add them to or remove them from the physics environment.

A special controller is the GameController, as this controller manages the running state and therefore the main game.

**GameController**
The GameController has several tasks to do for a game to run. Firstly it needs to create the game. This is done by loading a level, either from file or from a level factory and add this to a game object. Another task of the GameController is to update all entities contained in the game object. This means that all entities with state New must be added, those with state Alive must be updated and the entities with state Dead must be removed.
Updating entities is done by calling the update method on them.

**Entities**
Entity is the interface implemented by AbstractEntity that all entities in the game inherit. This class adds the concept of entity states to the drawable interface. This state is either new, alive or dead, representing an entity that is not yet added to the renderer, is already added to it or should be removed from the game respectively. By implementing the drawable interface each entity has a spatial that can be added to the renderer. Each entity can also choose to be a collidable object by implementing the PhysicsObject interface.

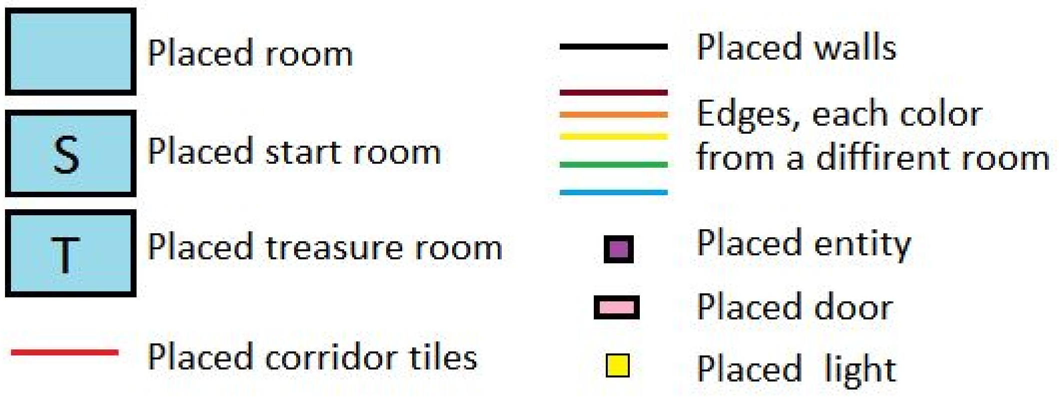
## 2.1.2 Overview of classes



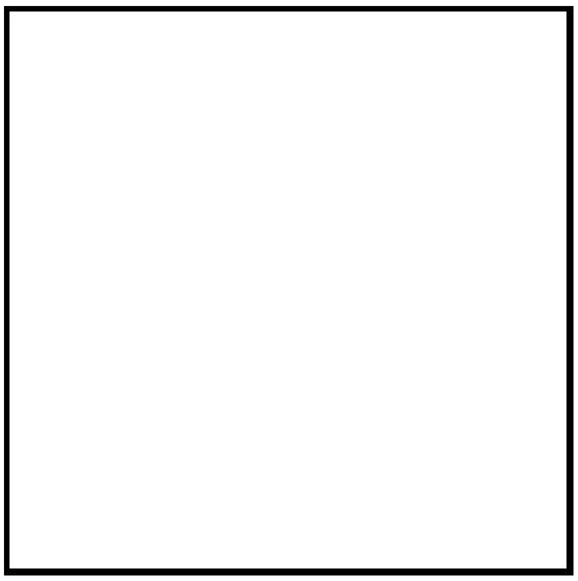
This is a quick overview of all game related classes in the program. Note that this is a very shallow UML diagram, only containing inheritance and the most important components of a class. The UML also shows the utility classes. Drawing all the dependencies between the main classes of the game and the utility classes would result in an unreadable UML, thus those dependencies are not shown. This image in a bigger resolution can be found on github in the documentation folder.

## 2.1.3 Maze generation

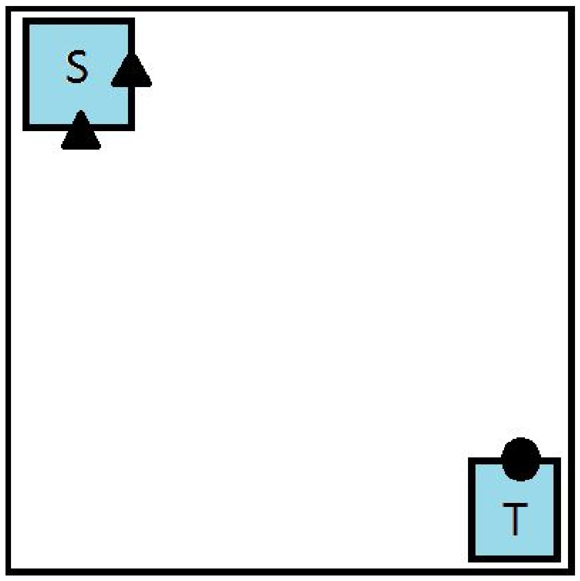
The maze generation is a large part of core gameplay experience of the game. When a game is started the maze generation runs and creates a playable level. Rooms are loaded from files, so new rooms can easily be added. The algorithm behind this uses a minimum spanning tree algorithm and a breadth first search algorithm, combined with an exponential distribution. To make this more clear the algorithm is explained in a set of images explaining each step. Firstly a legend is given and afterwards the set of images is given.



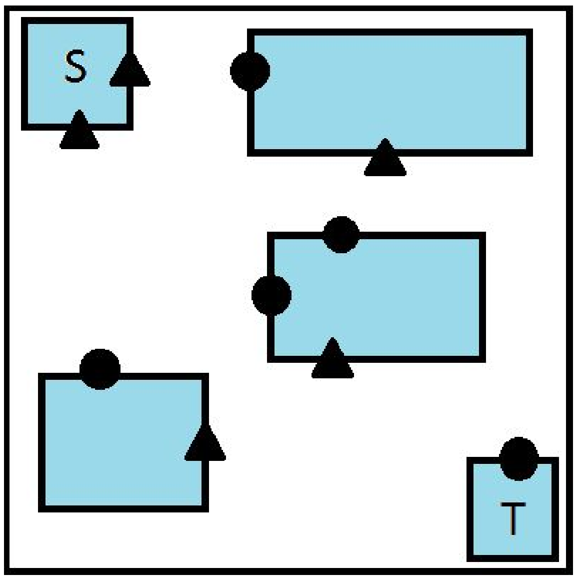
Legend of the maze generation algorithm.



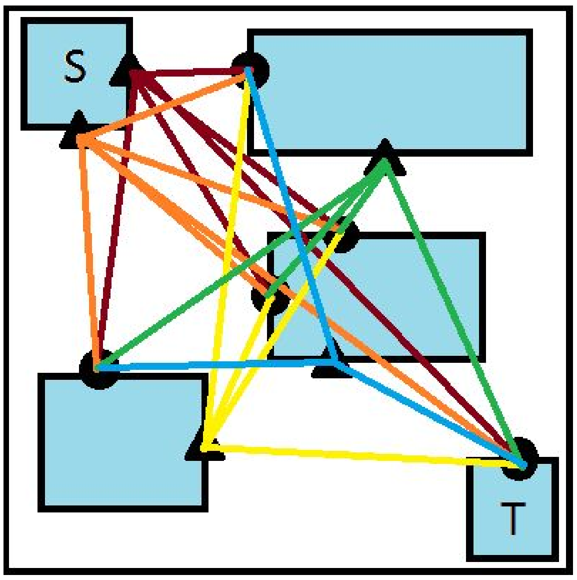
First create a playing field in which all maze tiles will be placed.



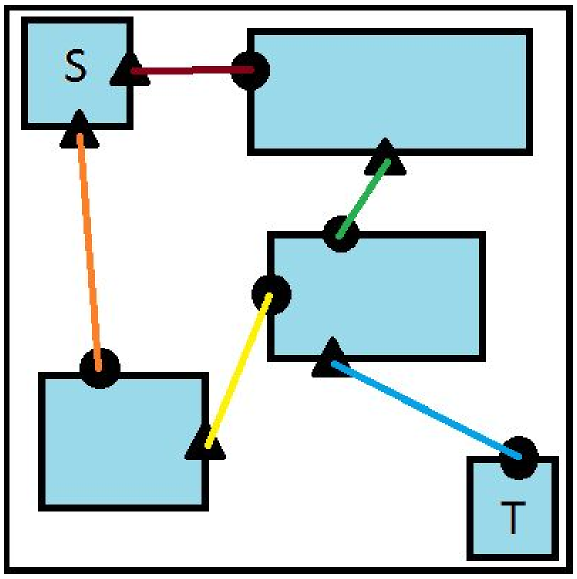
Place the start and treasure room, as these rooms must be present to create a completable maze



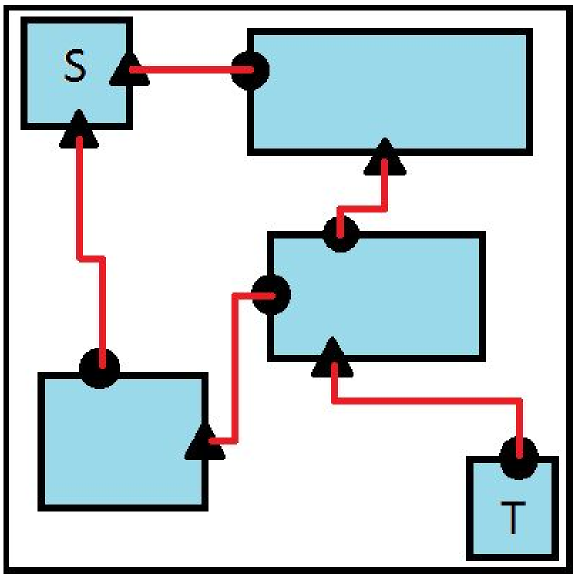
Place the other rooms from a list of all all the available rooms. Duplicates can be turned on. This step will try to place the rooms randomly until a maximum amount of attempts has been reached or all rooms are used.



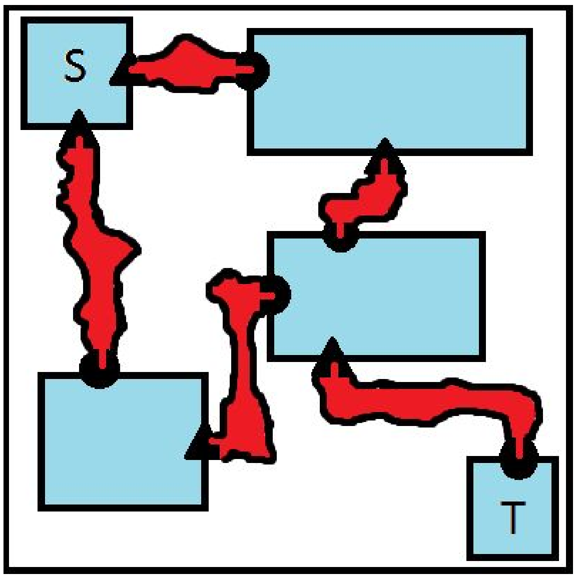
Connect each room exit to all room entrance of other rooms with an edge. This edge is not yet filled in on the playing field, as the number of total edges have to be reduced as much as possible.



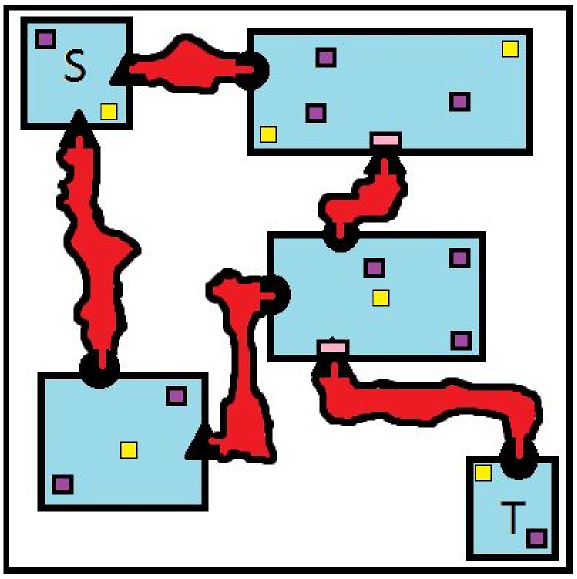
The maze generation is put through a reverse delete algorithm, which keeps deleting edges. It ensures that the every room is still reachable from an exit and every exit is still connected to at least 1 entrance. This extra constraint violates the principle of a minimum spanning tree, but ensures that the maze is playable.



Each of the remaining edges hold a start point as a room exit and an end point as room entrance. Because these are locations in the maze, a breadth search algorithm can be used to find the shortest route through the maze on the existing maze.



Corridors are widened using an exponential distribution to create more cave like structures. After that walls are added to the corridors, enclosing the maze entirely.

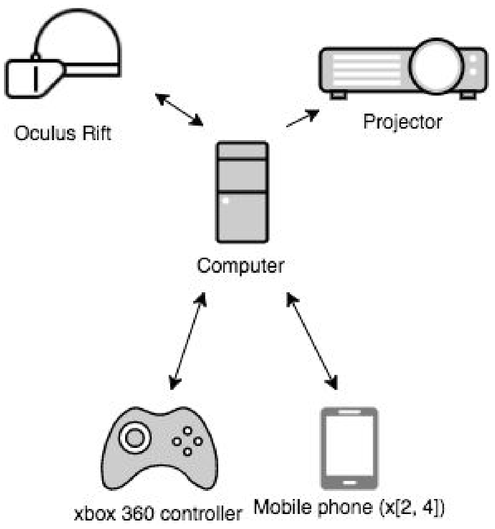


Entities are added to map, this includes enemies, traps, void platforms, gates, doors and much more. Lights are also added to the rooms. They are automatically turned into torch models and rotated correctly to attach to the walls.

## 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.

When a client connects, the server will send them the index.html page. Everything that happens next is determined by the client side. The client side has different views, depending on the state of the server. Initially, the client will be shown the index view. They can then choose to view the instructions or to join the game. Joining the game will only be possible when it is not yet full and has not started yet. The client will update its page every five seconds to disable/enable the join game button, and to check if they should switch to a different view.

When the client presses the join game button, a message is sent to the server. The server can then allow or disallow the request. If allowed, the server responds with the current game state, a cookie which will be used for identification, and will create a websocket connection with the client. The client will then switch to the view that is linked to that game state. For example, if the game is currently in the tutorial state, then the client will be shown the tutorial as well.

All clients that are connected to the server will be sent status update messages every 150 milliseconds. A status update message consists of the current game state, the client's team and state specific information. In the tutorial and paused states, no additional information will be sent. In the waiting and playing state, the entities will be added. For elves, explored tiles are also included in the message. In the ended state, the winner of the game will be added in the message.

Upon receiving a status update message, the client will:

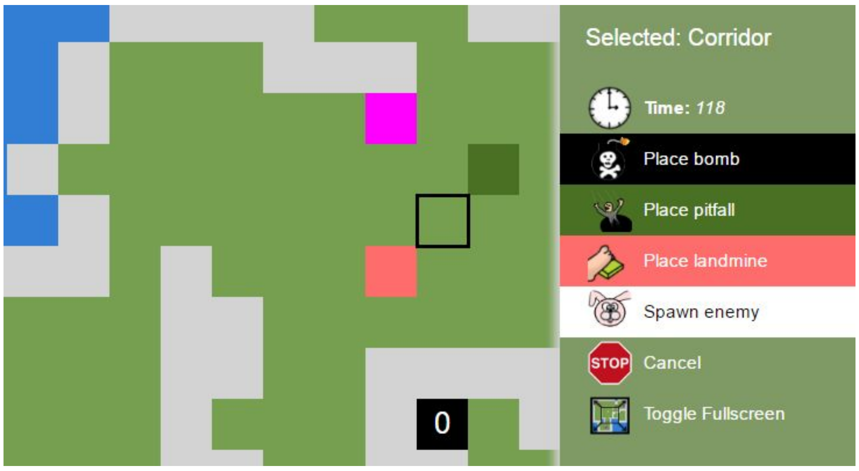
- Ensure that the view corresponding to the received state is being displayed.
- [Menu + Playing] Update the map with entities and exploration
- [Tutorial] Display the client side tutorial
- [Paused] Show a message that the game is paused.
- [Ended] Show a message displaying if the user won or lost, depending on the team in the message.

If a client drops out of the game at any point in time, they will be immediately put back into the correct state when they reconnect. The server uses the client's cookie to determine if they are in the current game or not, and if so, will create a websocket connection with them after the page has been requested, immediately starting the periodic status updates. The client will switch to the appropriate view and can continue playing as if nothing happened.

When a client performs an action, they will send an action request message to the server via the websocket. The server will either execute the action, or send an error message with the reason why the action could not be executed. No reply will be sent when the action was successful, as the next status update will include the effects of the action if it was successful.

**Playing View**

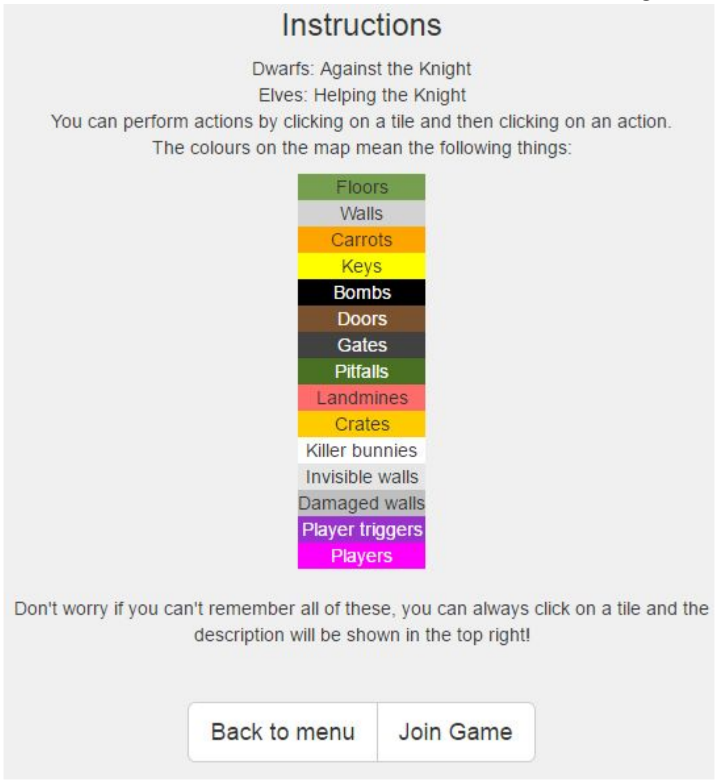
When the game is in the "playing" state, the client will be shown the map of the game. When the client first switches to this view, it will request the map from the server. Map information includes the layout of the maze (floors, walls, void). Only tiles that are also marked as explored will be actually shown to the client. Players can select a tile to see the action menu. (See the image below). In the menu, they can select different actions based on their team.



*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). 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).*

**Team selection view**
The team selection view is very similar to the playing view. The only difference is that there are two buttons to choose your team at the top of the page, and a button to quit the current game.

**Instructions view**
Below is an image of the instructions view, which is displayed when the user chooses "instructions" on the index view, and when the knight is walking through the tutorial.



Instructions

Dwarfs: Against the Knight
Elves: Helping the Knight
You can perform actions by clicking on a tile and then clicking on an action.
The colours on the map mean the following things:

| |
|---|
| Floors |
| Walls |
| Carrots |
| Keys |
| Bombs |
| Doors |
| Gates |
| Pitfalls |
| Landmines |
| Crates |
| Killer bunnies |
| Invisible walls |
| Damaged walls |
| Player triggers |
| Players |

Don't worry if you can't remember all of these, you can always click on a tile and the description will be shown in the top right!

*The instructions in the web interface.*

**Non-websocket support**
When a client does not support the use of websockets, the communication takes a slightly different approach. Instead of the server sending updates every 150 ms, the client will request

updates from the server every second. When an action request is allowed, the server will respond with an empty response, instead of giving no response. All other communication happens in the same way as with websockets.

## 2.3 Persistent data management

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

- Rooms are stored in .crf files. CRF stands for Caerbannog Room Format, and is our custom format used for loading rooms. In these files each tile of the room is defined as well as the entities of the room. It is possible to place a compiled .class file to add custom interactions to a room, greatly expanding the possibilities of designing the room.
- Maps are stored in .cmf files. CMF stands for Caerbannog Map Format, and is our custom format used for loading maps. All rooms are placed in a single folder and linked in the CMF file by way of writing down the name. In the CMF file the start and treasure room have to be defined. The number of rooms has to be defined too. This allows for rooms to be excluded from the map generation without removing the actual files.

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. The webserver is the only multithreaded class. Every client has a websocket, which runs in it’s own thread, to guarantee a quick response. To allow clients to create entities, we use a Concurrent Hash Set for the entities. This collection allows adding elements from multiple threads without giving any problems.

## 2.6  Dependencies

Below is a list of dependencies, and the reason we need them/what they are used for.

| **Dependency** | **Reason** |
|---|---|
| Lombok | Adds convenient short ways to create try catch blocks (SneakyThrows), getters, setters, constructors, etcetera. Simply a useful developer’s tool. |
| Jetty | Embedded web server, which is used for the web server of our game. |
| Jetty Websocket | Enables us to use websocket with our web server. |
| jMonkeyVR | jMonkeyVR (and all of its dependencies) is the engine the game runs on. |
| JSON | JSON parser/writer, to easily work with the JSON format, for both room loading and the web server. |
| Kenglxn QRGen | For generating QR codes. |

# 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.

**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.