Architecture design

Team Kroket

10-06-2016

Irene van der Blij	ivanderblij	4385691
Jochem Heijltjes	jheijltjes	1534041
Mayke Kloppenburg	mlkloppenburg	4383265
Alan van Rossum	alanvanrossum	4293932
Harvey van Veltom	hvanveltom	4350073

Table of contents

1 Introduction
  1.1 Design goals
    1.1.1 Code quality
    1.1.2 Availability
    1.1.3 Performance
    1.1.4 Maintainability
2 Software architecture views
  2.1 Subsystem decomposition
    2.1.1 The Oculus Rift
    2.1.2 The Wireless Gamepad
    2.1.3 Desktop Machine
    2.1.4 Game Host
    2.1.5 Android Devices
  2.2 Hardware/software mapping
  2.3 Persistent data management
  2.4 Concurrency
3 Glossary

1 Introduction

This document will provide an outline of the system that is being built by Team Kroket during the Context Project in the context Computer Games. This chapter will elaborate the design goals of the system.

1.1 Design goals

The design of a system is very important for every developer working on it. Therefore this section will discuss the desired design goals for the system.

1.1.1 Code quality

In order to have a system that is easily altered, extended and understood, the system should have adequate code quality. The master branch should always be of good quality; this means that everyone will do their work on a separate branch. Continuous integration will be used, so whenever someone thinks the work on their branch is finished and the code quality is on the same level as the master branch, he or she can create a pull request. At least two people should review this pull request, not only looking at functionality, but also verifying the code quality. In this manner cleanness of the code in the master branch is ensured, and thus that the system that is released after every sprint has adequate code quality. The code should be neat, clear and as bug-free as possible. To this end, Checkstyle, PMD and Findbugs are being used.

1.1.2 Availability

In order to create a system that meets the requirements of the stakeholders, a system with high availability has many advantages. At the end of each week a playable release is made available to the stakeholders. This is done by using Scrum and weekly sprints. This allows the stakeholders to give feedback during the development process, and adjust their wishes and requirements. When there is the possibility to give feedback during the development process instead of only at the end, you can make sure that the product you are developing is what the customer/stakeholders want and require. In this manner, the stakeholders also have a better overview of the product they are going to receive at the end, so they will not be surprised (or disappointed).

1.1.3 Performance

In every computer game, performance is of great importance. Game players do not want to wait long amounts of time for the game to respond; they want to keep on playing. This also needs to be incorporated into the to be build system. The performance of the system will also depend on the actual hardware that is going to be used. To ensure high performance on all machines, our game has the possibility to change certain graphics settings, such as screen size and shadow map size.

1.1.4 Maintainability

For a system to be maintainable it should be easy to modify the system and to correct faults. It will also ease improving the performance of the system and make it easier for the system to adapt to another environment. Iterative development and regular feedback, as mentioned earlier, already contribute to the maintainability. The code of the system should also be

readable and easy to understand. This can be achieved by documentation and adequate comments explaining the code.

2 Software architecture views

This chapter will describe the architecture of the system. The first section is about the subsystem decomposition, the subsystems and dependencies between them. The second section is about the hardware/software mapping. In the third section the database and the database design will be discussed. The last section is about concurrency.

2.1 Subsystem decomposition

The architecture can be divided into different subsystems. As of now it consists of 5 different subsystems. The Oculus Rift, The Wireless gamepad, two android devices, a game host and a desktop machine. A high level overview of the different subsystems and the interactions between them can be seen in figure 1.

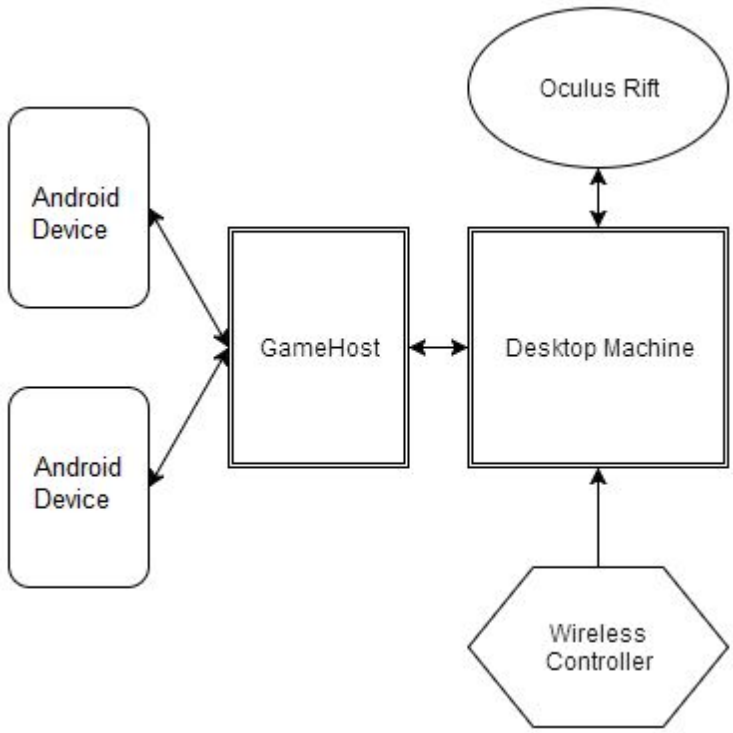

Figure 1: Architecture Design of the game, the shapes mean it is the same type of hardware

2.1.1 The Oculus Rift

The Rift will interact with the Desktop Machine. The desktop machine will sent the Rift the image of the virtual world so the user can look around in said virtual world. In turn the Oculus Rift will send back the coordinates of the headset so the view can be updated accordingly. For example if the user in the Oculus Rift looks left the new coordinates are sent to the Desktop Machine and in turn the updated image of the virtual world is sent back to the Rift.

2.1.2 The Gamepad

The gamepad does not have to be wireless. The gamepad is a way for the oculus rift user to send input to the desktop machine. For example if the Rift user needs to inspect an element he can press a button on the gamepad. Then the Desktop machine will know what the Rift

user wants and will respond accordingly. Additionally, the AXYB buttons on the gamepad can be used to enter color sequences from the VR client side.
The gamepad is only used to send input to the desktop machine, it doesn't receive any kind of input itself.

2.1.3 VR client

The desktop machine interacts with several subsystems. Namely the Oculus Rift, the wireless gamepad and the game host. The desktop machine receives input from both the wireless gamepad and the Oculus Rift. The desktop machine will check the input of both and will update the image of the virtual world and send it to the Oculus Rift based on the coordinates it received from the Rift.

Besides these interactions, the desktop machine will also interact with the game host, by sending and receiving messages. The messages received from the game host will be interpreted by the desktop machine, and then the desktop machine will alter the environment of the Oculus Rift player accordingly. An example of this is that when the game should start, the desktop machine receives a start message, and then the room appears.

The desktop machine can also send messages to the game host. This will be done when the Oculus Rift player has executed certain important action that should trigger an event. An example of this is that the Oculus Rift player interacts with an object that should trigger a minigame.

The use of the managers in *figure 2* is simple; they manage different aspects of the game, e.g. audio or the GUI. The screens are all overlays with text, and the scene is the escape room with all its objects. Further responsibilities are in *figure 2.* The EscapeVR initialises all managers. The MinigameManager keeps track of the current minigame and starts and stops it. The minigame classes contain the logic for the minigames, so the VR client can play them.
The InputHandler handles all input from the gamepad. Currently there is also support for a keyboard.

The program running on the desktop machine has a lot of functionalities and responsibilities. An overview of the current software that runs on the desktop machine can be found in figure 2.

The VR client has an EventManager which we use to distribute events throughout the entire client. An object can be registered with the EventManager as a listener. Anywhere throughout the client, an event can be sent to the EventManager. Each registered listener will receive each event so it can deal with the event accordingly.

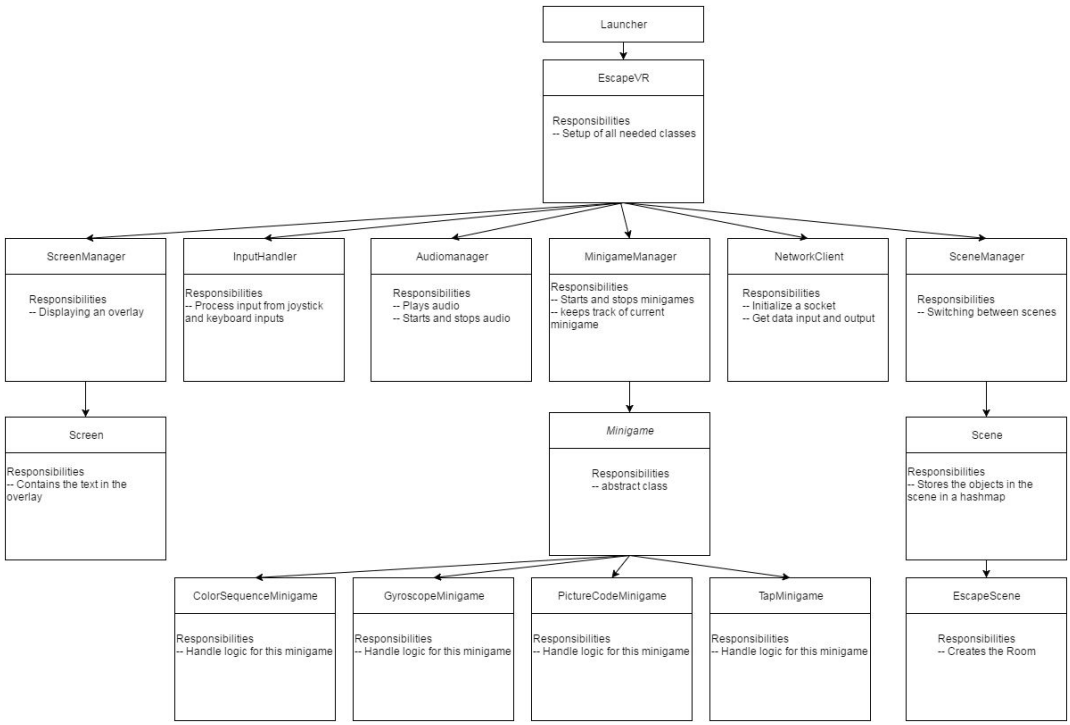

Figure 2: Overview of EscapeVR

2.1.4 Game Host

The game host interacts with both the Desktop machine and the android devices; it will allow both parties to communicate with each other. The desktop machine and android devices will send information to the game host. This information is about the status of the game. Whenever one Oculus Rift player and two mobile players are connected to the game host, the game host is responsible for sending a start message to all three players.

The subsystems are then responsible themselves for acting upon the message. The game host will also be receiving messages from the Android devices and desktop machine. Such a message can for example be that the Oculus Rift player has interacted with an object that should trigger a minigame. When the game host receives such a message, it is responsible for sending a message to the mobile players that they should start the corresponding minigame. When the mobile players have finished the minigame, the game host will receive a message from them and again forward this to the desktop machine, so that interaction between the mobile players and Oculus Rift player is possible.

A more detailed overview of the game host and its connectees can be found in figure 3 and 4. Figure 3 also contains the messages which each component can send. These messages are explained below.

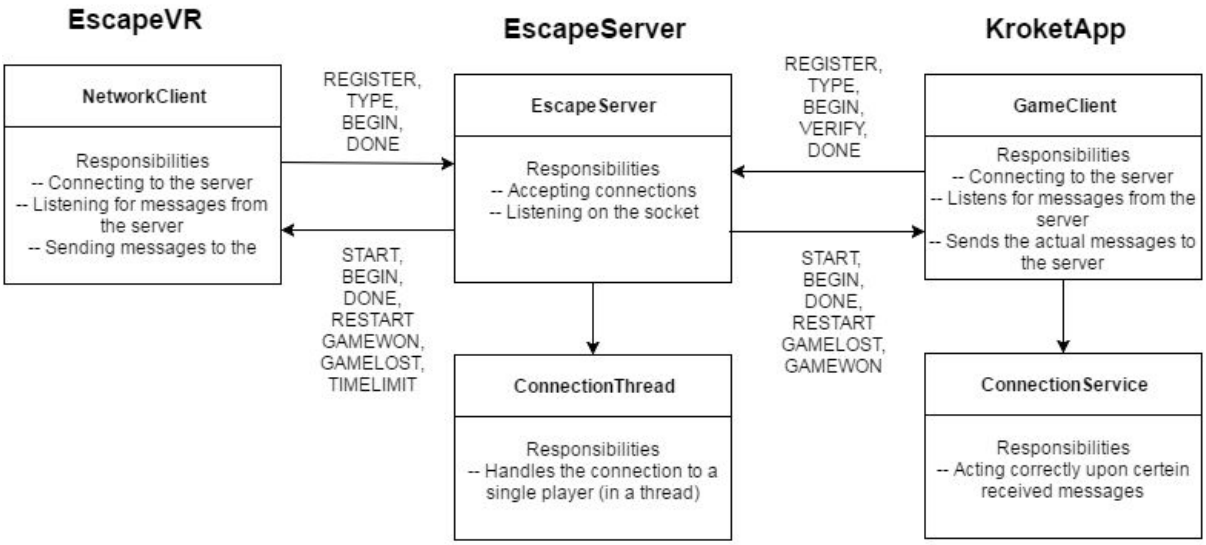

Figure 3: Overview server

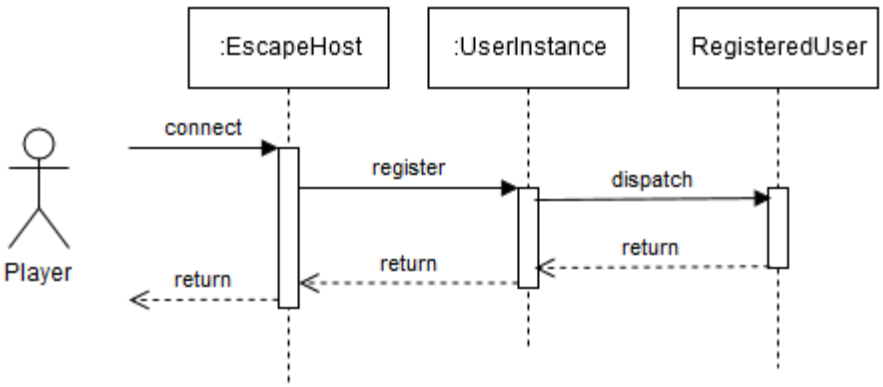

Figure 4: Sequence diagram server

As mentioned before the three components communicate with each other. Initially both the VR Client player and the app players will send a REGISTER and a TYPE message to the server. The REGISTER message contains the name of the player, and the TYPE message tells the host whether it is dealing with a smartphone player or an Oculus player. If all players are registered, the server sends all player a START message. This starts the game.

When the Oculus player interacts with certain objects, a minigame has to be triggered. The VR Client will send an BEGIN[x] message, where x is an identifier for which minigame, e.g. A or B. The server will change its gamestate accordingly and forwards the BEGIN message to all clients. When the clients receive a BEGIN message, the minigame starts.

When a game is finished, the clients will send a DONE[x] message, where x is the identifier for the finished game. Since some games require both smartphone players to complete the game, the server gets VERIFY messages from them. If the server does not get both VERIFY messages, or DONE from the VR client is not received, it sends RESTART to all clients. If the server gets both VERIFY messages and the DONE message from the VR client, it sends the DONE message to all clients. Minigames that only require one type of player to solve the minigame, only use the DONE messages.

The overall time limit for the all players is tracked in the server. The server sends TIMELIMIT messages to the VR client. When the server notices that the time is up, it sends a GAMELOST message to all clients. If the VR client has completed the last minigame, a GAMEWON message is send to the server, which forwards it to all clients.

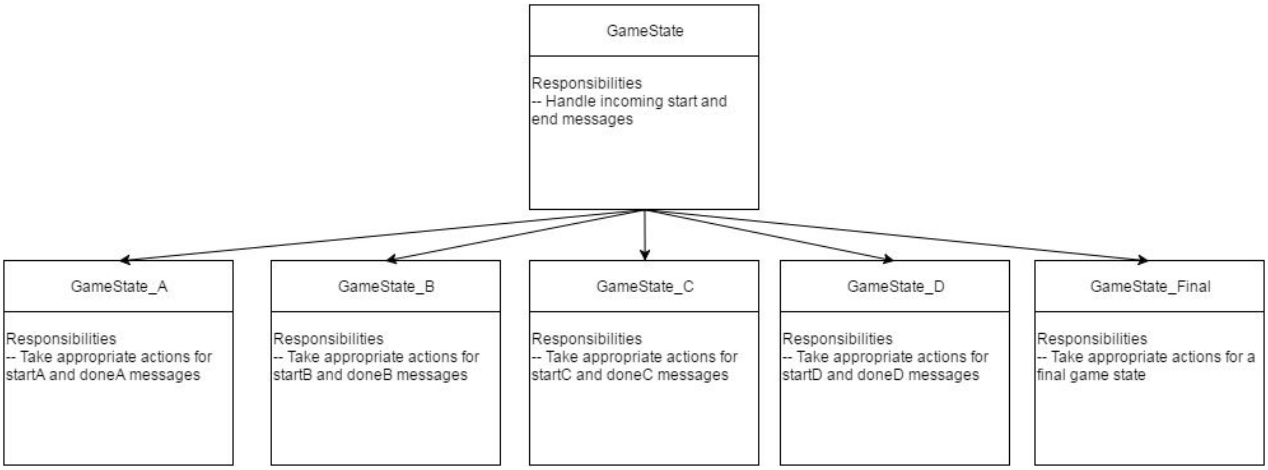

Figure 6: Overview of GameState classes

Also added are game states. Above is an overview of the classes. The GameState class is a singleton and has subclasses for all the game states. Initially, the singleton is an instance of GameState_A. When e.g. a DONE[A] message comes in, the instance is set to GameState_B. With each done message the GameState is instantiated to the appropriate subclass.

2.1.5 Android Devices

As already mentioned above, the android devices will interact with the game host by sending and receiving messages. These message will provide input for the game, and thus will trigger process. The messages received by the android devices are messages that are meant to let the device know which minigame to start. The messages sent by the android devices will contain the progress of the mobile players. A message will for example be sent when a minigame is finished successfully.

For every different screen in the android app there is a separate native Android activity class that is responsible for showing this screen. These classes also define what should happen when, for example, buttons are clicked in this screen. Besides the different activities there is a Service class (ConnectionService) and a normal class, which are already discussed above, and can be viewed in figure 3 and 5.

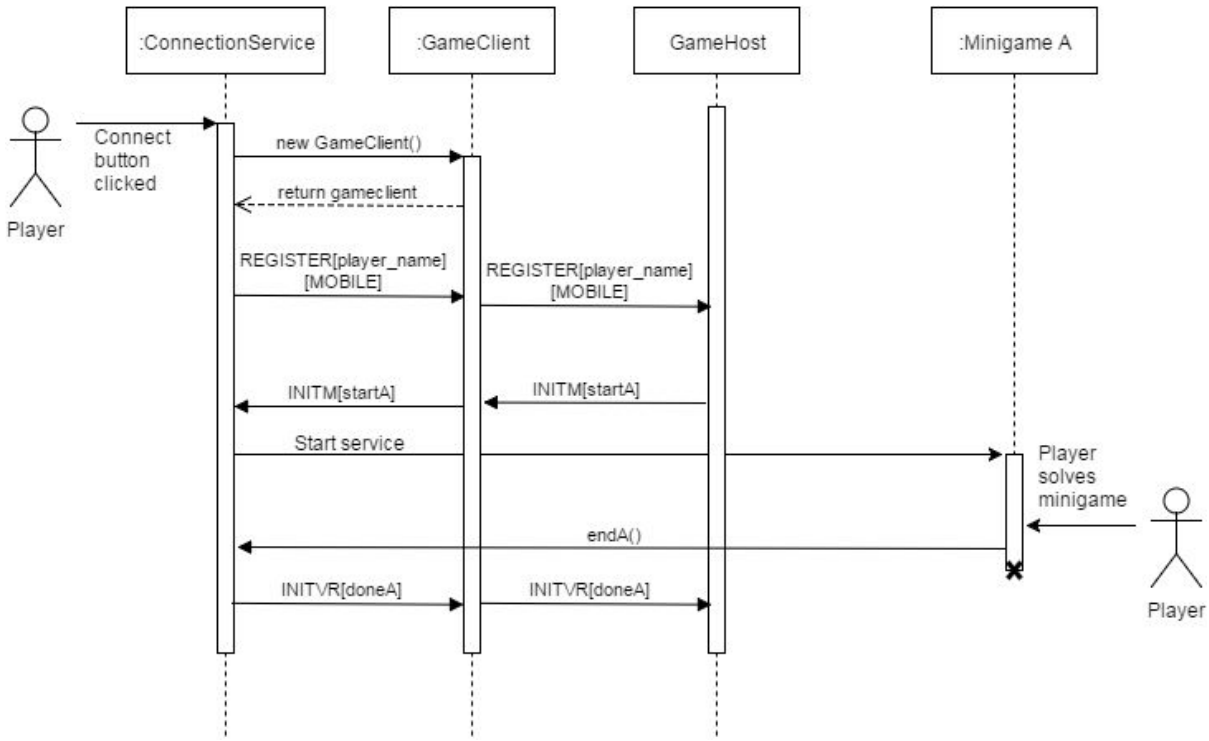

Figure 6: Detailed overview client side of the Android application

When the user start the app, a screen appears where a name has to be filled in. When this has been done the user can click the ‘connect’ button. This will connect the app to the EscapeHost. If the connection is established, the user can click start. The user will get a waiting screen until the Oculus player initialises a minigame. In the waiting screen, the mobile players can also play two minigames.

2.2 Hardware/software mapping

The software and hardware are different for the users, as illustrated in figure 6. When the game is started, a start screen will appear. In this screen, the player will be able to see his connection to the game. The Oculus Rift player will also see when mobile players have connected.

There is one Oculus Rift user. This user has the game on his computer. He can look around in the environment (the interface) with the Oculus Rift and move around and interact with the environment using a wireless gamepad. The other two users have an app on their smartphone. The smartphones and computer are connected to a server, so there can be communication between the Oculus user and the smartphone users.

A single application running on a desktop computer operates as host. The clients mainly communicate about whether a minigame on the smartphone can be initiated and whether a minigame is finished. All the clients interact with this host through TCP. For the game, it is necessary to exchange string messages to initialise events in the app and Oculus game. This means it is very important to have a reliable protocol, so it is certain messages have been received. This is why TCP is used for this project.

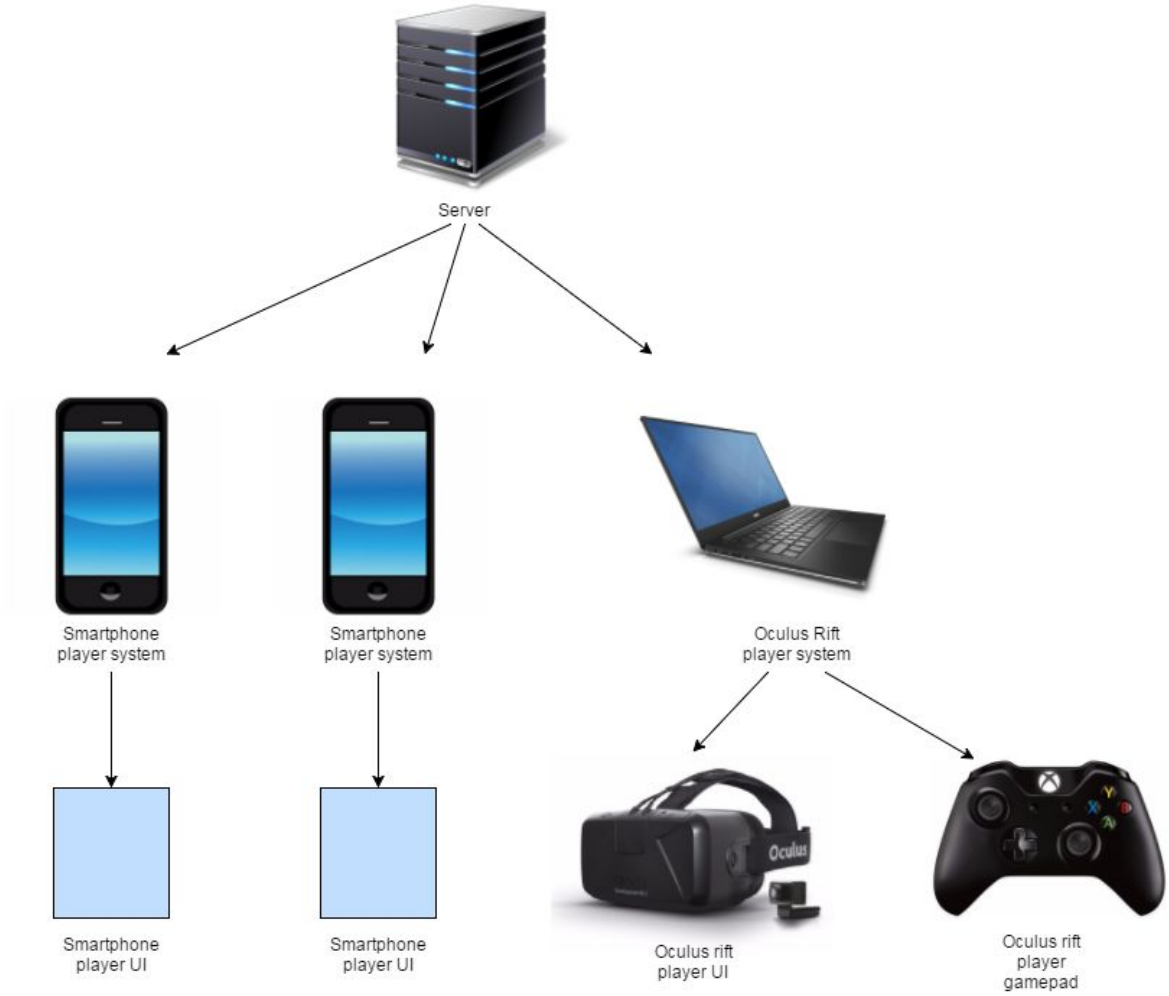

Figure 7: hardware/software

2.3 Persistent data management

The game concept does not require user data to be stored. It is a one time game without user profiles etcetera. Therefore, there is no need for a database design.

However, our system could have a form of high scores in the future. The scores would have to be stored throughout different sessions and thus require a persistent form of a database for data storage. The use of an actual RDBMS would be unnecessary, as a simple JSON or XML file would suffice. A simple method would read the data and output the 3 or 10 teams with the highest scores when prompted.

2.4 Concurrency

The system consists of multiple clients interacting with a gamehost, through TCP. Because the gamehost/server will be responsible for hosting an entire game session and ordering each client what to do, all data will flow through the server. The clients won't communicate with each other directly. The host is also responsible for using shared resources. We chose TCP over UDP for now because we want our data transmissions to be reliable and all we are exchanging is event.

In the gamehost/server, the accepting part is in its own thread. For each incoming connection, a separate thread is spawned so that each player/user is dealt with in its own instance.

Each connected client has to wait for incoming data from the server, such as events. To make sure the clients can keep running while waiting for incoming data, the receiving part of each client is dealt with in its own separate thread.

Because threading is potentially dangerous, we have to be extra careful when dealing with shared variables and incoming data. We are using synchronized methods and sharing only variables that can actually be shared between threads. A call to a method that uses resources only available in the render thread cannot be used in other threads. We are still considering using atomics/locks/mutexes or semaphores where using shared variables appears to be problematic or dangerous.

3 Glossary

Android - Operating system for Mobile Phones and tablets. Makes use of the Linux Kernel and the Java programming platform.
Branch - A parallel version of a repository. A branch is contained within a repository, but does not affect the primary or master branch.

Client - An application that relies on a server or host to perform operations.
Gamepad- An input device primarily used to play video games on a console, like Xbox or Playstation.
Host - A computer system that can be accessed by clients at a remote location, often through the internet.

Master branch - The default and main branch. Should contain the latest working version of the system.
Oculus Rift - A pair of Virtual Reality goggles with a stereoscopic field of view, designed for gamers. The device measures the movements of its wearer.
Pull request - A proposed change to a repository, submitted by a user, and accepted or rejected by the repository's collaborators.
RDBMS - Relational Database Management System, a database management system based on the relational model. Easy to understand choice of storing record data.
Repository - Git's name for the 'main folder' that contains all project files and each files' revision history. Collaborators can contribute to the state of the repository by committing and pushing changes to the project files.
Smartphone - A phone with many features running an advanced operating system, similar to a desktop computer.
Socket - An endpoint of a connection in computer networking.
SQL - Structured Query Language, a standardised language that's used to ask and alter data from a database management system (see: **RDBMS**).
Virtual Reality - An artificial world, generated by a computer.
TCP - Transmission Control Protocol, often used to establish connections via the internet.
UDP - User Datagram Protocol, a stateless/connectionless protocol for exchanging data over networks.