CHALMERS

A Flexible Platform for Wireless Local Multiplayer Gaming Using Smartphones as Controllers

Bachelor of Science Thesis in Information Technology

Gustav Alm Rosenblad
Lukas Borin
Pontus Pall
Emil Åsberg

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, May 2014

A Flexible Platform for Wireless Local Multiplayer Gaming Using Smartphones as Controllers

Examiner: Jan Skansholm

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden May 2014

Abstract

The purpose of this thesis is to give an in-depth technological view of a multiplayer video gaming system designed to let players use their smartphones as controllers. The system consists of two interconnected parts: a program on a personal computer that acts as a game console, and a smartphone application that turns phones into game controllers, enabling the user to interact with a game in new and creative ways.

Using smartphones as controllers allows each game to use its own, specialized control interface. The system facilitates this by employing remote code execution to send graphical interfaces from the server to the connected phone clients. This in turn creates a basis for a cross-platform system where the mobile parts of the games can be written entirely in a platform independent scripting language.

In order to evaluate the functionality and advantages of the multiplayer video game system, four games with different types of user interaction were designed. The use cases of these games provide practical evidence for the usability and applicability of the system.

Sammanfattning

Syftet med denna rapport är att ge en fördjupad teknisk överblick av en spel-plattform avsedd för flera användare som låter spelare använda sina smartphones som kontroller. Systemet består av två sammankopplade delar: ett program på en dator som agerar som en spelkonsol, och en smartphone applikation som förvandlar smartphones till spelkontroller, vilket ger användaren möjlighet att interagera med spel på nya, kreativa sätt.

Användandet av smartphones möjliggör skapandet av spel-specifika kontroll-gränssnitt. Systemet underlättar detta genom att använda fjärrkörning av kod för att skicka grafiska gränssnitt från servern till de anslutna smartphone-klienterna. Detta skapar i sin tur grunden för ett plattformsoberoende system där smartphone-delen av spelen kan skapas enbart med hjälp av ett skriptspråk som kan köras på flera operativsystem.

För att utvärdera spel-plattformens funktionalitet och fördelar designades fyra spel med olika typer av användarinteraktion. Användingsområdena av dessa spel ger praktiska bevis för plattformens användbarhet och tillämpbarhet.

Table of contents

1. Introduction **1**
1.1 Inspiration and motivation 1
1.2 Ethics and sustainability 3
1.3 Scope and contributions 4
2. Related work **4**
3. Design and Development **5**
3.1 Designing a platform 6
3.1.1 Remote code execution 6
3.2 Server development 8
3.3 Client development 9
3.3.1 Android Native Development Kit 9
3.3.2 Android Activity Lifecycle Management 9
3.4 Efficiency and Power Consumption 10
4. Platform **12**
4.1 Network 12
4.1.1 Client-server model 12
4.1.2 Bluetooth versus Wi-Fi 13
4.1.3 Protocol 13
4.1.4 Server Discovery 14
4.1.5 Connection 15
4.1.6 Disconnection 18
4.1.8 Content transmission 23
4.1.9 Latency 24
4.2 Serialization using the Simple Data Language 26
4.2.1 SDL Usage 28
4.2.2 Annotation-based extensions 29
4.3 Graphics 31
4.3.1 Images and text 31
4.3.2 Animation system 32
4.3.3 Particle system 33
4.4 Sound 35

4.5 Content reloading 35
5. Games 36
5.1 Achtung 36
5.2 Tower Defense game 37
5.3 Pictionary 40
5.4 Questionary 41
6. Future work 42
6.1 More platforms 42
6.2 User testing 42
6.3 RCE sandboxing 43
6.4 Resource version control 43
6.5 Lower latency network 43
6.6 Marketability 43
7. Discussion 43
8. References 46

1. Introduction

In today's western society most people have access to a smartphone [1] . While many use their smartphones for solitary activities such as surfing the web or messaging distant friends, surprisingly few applications allow individuals to use their phone to engage with others in the same physical space.

This situation is problematic for several reasons. Perhaps most noticeably, smartphones can serve as an excuse not to participate in discussion and other types of face to face interaction. An alarming example is the interaction and responsiveness between parents and their children, which might suffer due to excessive smartphone usage [2]. We aim to improve upon the situation by creating a virtual gaming console, where smartphones are used as the controllers and a computer acts as the console. This will allow smartphones to be used for social activities and might even strengthen the bonds of parents and children if they play together.

Obviously, a game console without games serves no purpose. Therefore, we decided to build several games to test the effectiveness of the console. With these games, we attempted to cover as much of the phone-computer interaction space as possible. We believe that this platform might appeal to many different audiences. Therefore, we tried to target different segments of the population in different games. The complete list of games created is:

1. **Achtung** - a fast-paced competitive multiplayer game, created with young people in mind.
2. **Tower defense game** - a cooperative multi-player game in the tower defense genre, targeted towards more experienced gamers.
3. **Pictionary** - a classic game where one player draws and the other players guess what the drawing portrays, aiming to entertain people of all ages.
4. **Questionary** – a simple trivia game in the style of Trivial Pursuit, aimed at adults and adolescents.

1.1 Inspiration and motivation

There are already products on the market that provide similar ways of controlling video games.

The Nintendo Wii U gamepad was the first major video game controller to include a touch screen, microphone, camera, accelerometer, and other hardware components usually found in smartphones. This controller makes it possible to create new kinds of local multiplayer games where information can be hidden between players. However, on the Wii U console only one such controller can be used per console [3]. This puts a limit on the kinds of games possible to make. For instance, in a game replicating the popular poker game “Texas Hold'em”; it would be preferable if each player could see their own cards on their own separate screen, hidden from the eyes of the other players.

An example of a game using hidden information is “Animal Crossing: Sweet Day”. This game is included in the Wii U title “Nintendo Land” and can be seen in figure 1.1. In this game, the player using the Wii U gamepad is playing on her own against all other players. The player with the gamepad chases the other players around. The interesting part is that the other players can

only see the chasing player's avatar when it is in close proximity to their avatars, while the chasing player can always see her avatar on her Wii U gamepad.

Figure 1.1: Animal Crossing: Sweet Day

A similar approach is adopted by Xbox SmartGlass [4], a companion application for the Xbox 360 and Xbox One video game consoles. Xbox SmartGlass allows players to control console applications using a smartphone or tablet. Consumers who watch YouTube videos on their Xbox can use the touchscreen keyboard of their smart device instead of having to type using the Xbox controller or a conventional TV remote. While in-game, players can enrich their experience by viewing additional in-game information such as a dungeon map or boss statistics on their smart device. In the racing game Forza Horizon the player's SmartGlass device can be used as a virtual GPS to navigate the game world. This can be seen in figure 1.2. Additionally, the touchscreen and other sensors can be used to interact with the game in new ways. For example, in the virtual baseball game Home Run Stars, a second player can pitch the ball using the touchscreen of a SmartGlass enabled device.

Figure 1.2: Forza Horizon

Unlike the Xbox SmartGlass application, our console makes the phone the primary input device, and unlike Wii U's gamepad that only one player in the group can use, our system allows each player to have their own screen. This opens up for new and more innovative ways for the users to interact with the games. Furthermore, people carry their smartphones with them at all times, ensuring there will be enough controllers available to meet the demand. Figure 1.3 presents our vision. The figure showcases a computer which displays graphical content on a shared screen and each player using a smartphone as a controller.

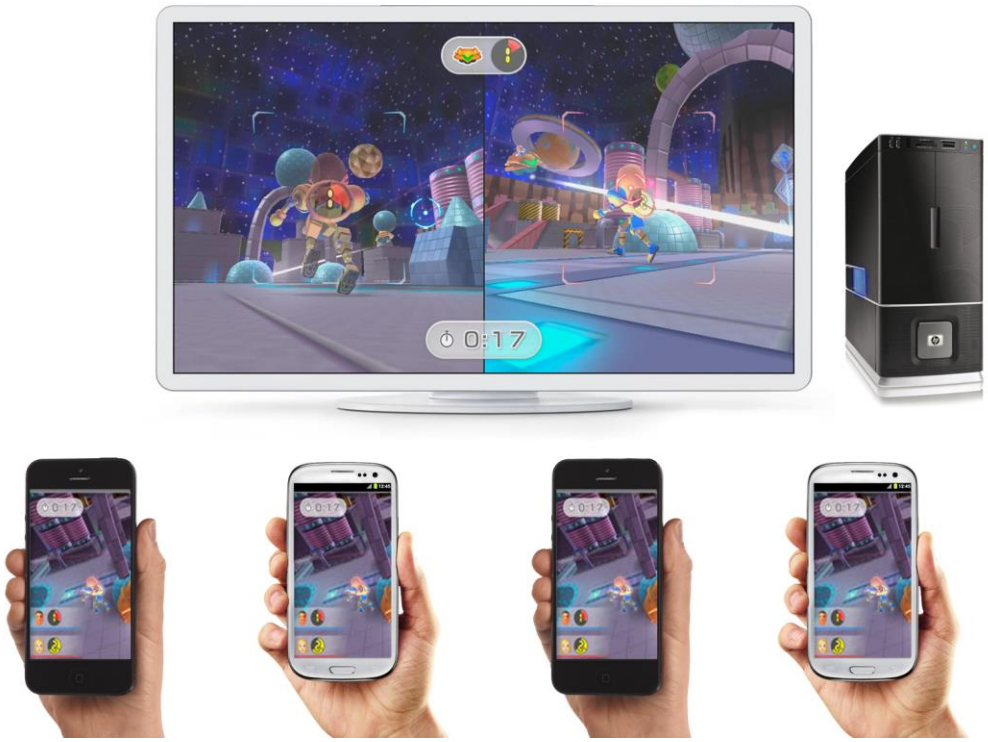

Figure 1.3: A game being played by four players using their smartphones as controllers.

1.2 Ethics and sustainability

In a society that grows increasingly individualistic it is important to engage people in social activities. Low levels of social interaction have been found to be as dangerous as being an alcoholic or smoking 15 cigarettes a day [5]. With this in mind all the games made for this platform are designed to be played by several players in an attempt to encourage collaboration and friendly competition.

The system is based on hardware that most people already own [6]. This means that no additional devices have to be manufactured, transported across the planet and finally purchased in order to use our system. Because of this, the console can be viewed as having close to no environmental impact.

The system can be run on phones which use a relatively old version of the Android operating system [7]. This implies that many older phones that have been replaced by their owners have the

opportunity to be useful once again, thereby extending their lifetime. This will in turn prevent them from being disposed of for a longer period of time, which will reduce the strain on the electronic waste disposal systems.

1.3 Scope and contributions

Smartphones and computers come in many varieties, with different hardware and software configurations. In order to serve a broad audience it is important to build a general platform that can accommodate as many devices as possible. The aim of this project is to build a proof of concept platform. Thus, we only consider the Android operating system on the mobile side and the Microsoft Windows operating system on the computer. However, in the future we want the system to be platform independent.

When designing a platform used for gaming purposes, end user experience is of utmost importance. Given this, games must be designed with the intention of keeping a stable frame rate and controller input latency has to be kept at an acceptable level. Another important aspect is power consumption. If the system is designed without concern for power consumption, the smartphones battery will be drained quickly.

The user should not have to suffer from the frustrating experience of updating the application every time a minor change to a specific game is made. Therefore, this process of updating should be as automatic as possible.

Given our focus on social interactions we will not create any single player games. Instead, we will focus on creating games which multiple persons can enjoy. These games will not have an set player limit and instead support any number of players.

This project has three main contributions. The design and development of the system (Chapter 3), the implementation of the platform (Chapter 4) and the games created for the platform (Chapter 5). Each of these contributions will be addressed in the following chapters.

2. Related work

This chapter describes similar projects which have been attempted elsewhere. New digital modes of interaction are a popular research topic in the realm of human-computer interaction (HCI). Unsurprisingly, projects similar, though not identical, to this one have been attempted by others.

Malfatti et al constructed a software platform for connecting smartphones via Bluetooth to a central computer [8]. After implementing four games using the technology, they concluded that the platform allows for a more natural way of interaction as compared to a keyboard and mouse. An interpretation of the classic arcade game Strikers 1945 built for the platform is shown in figure 2.1, with the phones acting as controllers being displayed at the sides of the image. The platform produced differs in significant ways from the one presented in this paper, but still serves as a proof of concept. The implementation relies on phones running the Symbian operating system, and is therefore unfortunately outdated to the degree of being completely unusable.

Figure 2.1: A clone of Strikers 1945 being played on two Symbian phones.

Many projects aim to let a smartphone act as a substitute for a traditional video game controller. This is done by mapping keyboard and mouse functionality to the touchscreen, accelerometers, and gyroscopes. One such application is Wi-Fi PC Game Controller [9], made by Pocket App Developers. This application allows players to control the computer keyboard using buttons projected on the touchscreen, and control the mouse using the touchscreen as a touchpad.

Joselli et al implemented a 2D shooting game controlled by smartphones connected to the central computer via Wi-Fi, and evaluated three different control schemes [10]:

1. Buttons projected on the touchscreen.
2. Shooting using a touchscreen button, steering using motion controls.
3. Shooting using a swipe gesture on the touchscreen, steering using motion controls.

From this evaluation, the conclusion was drawn that the two latter, more natural, options were considered superior by users.
Golomb et al and Burke et al have exploited the immediacy of gesture based interfaces in physical therapy, and have thereby demonstrated the broader importance of the topic [11] [12]
The basic concept has also been applied to public displays (for example displays found in libraries or school restaurants), since these seldom offer a high-fidelity method of input [13] [14] [15].

3. Design and Development

This chapter describes the design of the platform architecture, followed by an in-depth discussion of what libraries and languages are used, and why. Finally, a brief discussion of the efficiency and power consumption of the platform is presented.

3.1 Designing a platform

The design decisions that were made when designing the platform are discussed in this section. It contains the requirements of the system as well as a motivation for using remote code execution (RCE) and how it is realized on the platform.

There are several hardware components in a smartphone which make it an interesting input device for games. The touch screen, accelerometer and camera all present different opportunities in terms of user interaction. Thus, a functional requirement of the system is to be able to read input from a number of hardware types, including the ones mentioned above and more. The system also needs a way to present contextual information on the touch screen.

In order to cater to different types of smart phones, with different operating systems, the design of the system needs to be created with platform-independence in mind. To better accommodate the needs of the intended audience (attendants of a social event); it is of utter importance that the process of setting up and starting a game is as quick and simple as possible.

Given the requirements presented above, a basic system architecture can be created. A fundamental version of the mobile application needs to:

- Connect to the game console (in our case, the computer)
- Transmit input information
- Show graphical user interface

The first two items in the list concern network programming, which will be discussed in depth in section 4.1. However, the last point has direct implications for the fundamental design of the system. If the conventional way of building applications is followed, where a GUI has to be created separately for each mobile operating system, the system cannot be considered platform independent. To be able to provide the same graphical interface for more than one mobile operating system, a common platform must be built. But where does the graphical interface reside, if not on the mobile? A logical answer is the server, the only node in the network that all phones communicate with (see client-server model subsection 4.1.1).

3.1.1 Remote code execution

In this subsection we will discuss what remote code execution is and how the platform makes use of it. We also examine the advantages and disadvantages of using RCE.

The server is responsible for managing all game and controller logic, as well as the graphical interfaces for both the server and the smartphone controllers. The parts of the code that concern the smartphone controllers will (once a connection between the player's phone and the server has been established) be sent over the connection, received by the phone and executed directly afterwards.

This process, where code is first sent from one computer to another, and then executed, is commonly referred to as *remote code execution*, or RCE in short [16]. It is a fairly well-known method which can and has been used to perform potentially illegal actions in poorly secured IT

systems. There are of course legitimate, non-malicious, applications for RCE as well. A good example of this is online systems for learning new programming languages. The user in such a system is typically asked to enter some lines of code in the relevant language, which is then sent and executed on the server, yielding a result which is then sent back to the user. This makes it possible to get a hands-on experience with a language, without having to install a compiler, virtual machine or other additional software on the computer. An example of such a system is Try Clojure [17].

RCE presents several advantages from the point of view of the software engineer. Once the networking and code interpretation parts of the mobile application are in place, no further development has to be made on the mobile side. This makes the system easy and fun to work with for game developers, since they can focus all of their energy on building game logic and control interfaces, instead of battling OS-specific peculiarities. Another advantage that comes from automatic distribution of code is that any changes or additions to the games can be tested instantaneously, on several devices. This in turn yields far more rapid iterations compared to the usual cycle of compiling, installing and then rebooting the application. Since there is a central server that maintains and dispatches the only available version of the game logic (see 4.1.8), inconsistencies between different phones cannot occur.

From the user's perspective, RCE makes the experience of using the mobile application a lot smoother. Since the games are delivered by the server, there is no need to update the application to play new games. The only exception would be if some new hardware support was to be added to the underlying system. The threshold of getting started is reduced further due to the fact that the application mostly acts as a simple input device, and thus is minimal in size.

A number of issues have to be resolved before a system using RCE can be considered functional. First and foremost, it must be decided what language to use for the remotely executed code. This decision has large implications for the underlying platform used to interpret and execute the code on the receiving end. Since we want the code to run and interact with the hardware on both Android and iOS, the chosen language needs to be interoperable with C/C++. Additionally, the language should be performant enough to never suffer any noticeable slowdowns during gameplay. In order to guarantee this, garbage collection must be either non-existent or incremental.

On the basis of these criteria, we settled on the programming language Lua. Lua is a dynamically typed language intended to be used as a scripting language [18]. Due to this, Lua is compact and easily embeddable. The specific Lua implementation chosen was LuaJIT, since it is arguably faster than all other Lua implementations [19] and runs on most platforms [20].

Lua is a minimalistic and lightweight language. This makes it easy to generate Lua code. The language syntax fits on a single page [18] and is very easy to learn. It is often used by non-programmers such as artists and designers on a game development team. It is possible to sandbox code on a file by file basis in Lua, in order to safely make use of RCE. However, sandboxing code is not included in the scope of this project.

3.2 Server development

In this section, we describe what languages and libraries we used to build the server implementation. Subsequently, the benefits and problems associated with a key feature of the D programming language, compile time reflection, is discussed.

We use the D language in combination with a representational language of our own making, the Simple Data Language (SDL), for all server development. We chose D since we appreciate the native control, the fast compile time, the platform independent standard library and the powerful compile-time computational abilities it offers [21]. The compiler used was the Digital Mars D compiler (DMD), which is the reference compiler for the D programming language [22].

To interact with the hardware we used the Derelict binding libraries [23]. These libraries contain bindings to many popular platform independent game oriented C libraries. In combination, these libraries provide most of the core functionality required to make any 2D game imaginable. When selecting the libraries we valued quality of implementation, platform independence and documentation. The selected libraries are all very strong on all three criteria. The libraries we used were:

- GLFW - windowing library [24]
- FreeImage - image loading library. [25]
- SDL_mixer - sound playing/mixing library. [26]
- OpenGL 3.3 - low level graphics library. [27]

The D language has very powerful compile time reflection capabilities. This is utilized to great effect in many places in the codebase. An example of this is the automatic serialization of messages which is explained in subsection 4.1.7. Another example is the SDL parser that makes use of annotation based reflection to parse SDL code into D code. This whole process is described in subsection 4.2.2. These reflection capabilities provide the programmer with the ability to write fast code with clean interfaces.

We did run into some problems while developing low level reflection based code. These problems were at times caused by bugs in our code, which were evaluated at compile time, and in some instances by bugs in the DMD compiler itself. When bugs appeared at compile time they usually involved the compiler crashing with some unreadable error message emerging from deep within the compiler implementation. Normally these bugs were related to incorrect forward declarations. However, the errors did not give any indication as to what the problem might be. This meant that whenever this happened, it would take a while to figure out what was wrong and then either fix the code or work around the compiler bug.

Partway through the project an update to the DMD compiler arrived. This update broke some of our existing code. However, this code break was due to the usage of deprecated functionality. A lesson learnt from this experience was that when coding in the D programing language, one should avoid using features that are deprecated, since these might be removed in the next compiler update.

3.3 Client development

This section gives a high level overview of the Android application. It begins by detailing which languages are used. This is followed by an introduction to the Android Native Development Kit, which is in turn followed by an overview of a difficult problem - dealing with the Android lifecycle.

The Android client is written in Java, C/C++ and Lua. The reason we decided to include C/C++ and Lua, and not simply write the entire client in Java, was that we wanted as much code as possible to be easily portable to the iOS operating system. C/C++ is natively supported by iOS and is supported by Android through the Android Native Development Kit (NDK) [28]. As discussed in section 3.1 we use Lua to extend the client.

3.3.1 Android Native Development Kit

The NDK makes it possible to write Android applications in the C/C++ programming languages. It provides access to many common low level libraries such as the OpenGL ES 2.0 graphics library [29] which is used as the rendering library on the client. However, it is not possible to access all the features the Android operating system provides. An example of a feature that is not present in the NDK is the Android GUI. If any Java feature is required it has to be accessed through the Java Native Interface (JNI) [30]. JNI is a C/C++ library that makes it possible for C/C++ to access code written in the Java programming language and for Java to call annotated C/C++ functions.

The Android NDK is not as user friendly as the Android Software Development Kit (SDK) [31] which is normally used to build Android applications. This seems to be by design and Google makes it clear on the NDK download page [28] that one should not use the NDK unless it is required, which in our case it was.

A challenge of working with the NDK is the lack of documentation. The NDK does contain some HTML help files. However, they are usually not up-to-date and thus contain many errors. When we started working with the NDK this was of course a huge problem. But over a couple of weeks of technical difficulties we managed to implement most of our features. We were able to achieve this by studying the different samples included in the NDK. We also studied the NDK implementation itself and investigated how to use it by applying the technique of trial and error. One problem that is yet to be overcome completely is the Android lifecycle management.

3.3.2 Android Activity Lifecycle Management

One of the hardest design and implementation problems encountered throughout the project was management of the Android activity lifecycle. The Android life cycle is complex. It has over a dozen important events that occur in different order and in different quantity depending on what phone is used and how the user interacts with the phone. A detailed guide over of many of the peculiarities in the Android lifecycle is shown in a white paper from Nvidia [32]. This resource proved to be invaluable in implementing the system. Many of the corner cases could successfully be avoided and in the end the client reached a fairly stable form. This is still a work in progress and will require some additional testing to achieve complete stability.

3.4 Efficiency and Power Consumption

In this section we discuss various measures taken in order to improve the performance and efficiency of the client and server applications. We also describe a custom sleeping algorithm that was developed and how this helped reduce power consumption without negatively affecting gameplay.

Games are generally considered to be soft real-time applications. That is, the usefulness of a computation result degrades after a deadline. There are very tight constraints on how long a game can do computation in a single frame. In the typical case these constraints are either 16.7 milliseconds for a 60 frames per second (FPS) game or 33.3 milliseconds for a 30 FPS game. Our games run at either 60 FPS for animated/low-latency games (Achtung and Tower Defense) or 30 FPS for games with less emphasis on graphics (Pictionary and Questionary).

While neither the client nor the server has any problems rendering 60 or 30 FPS, they both suffered from a power consumption problem. After the necessary computation had been performed for a frame of gameplay, the game would wait for the monitor to refresh (this event is often called vertical synchronization, or vsync for short). However, instead of letting the processor sleep this would consume a full CPU core which would consume a lot of power. This was resolved by allowing the processor to sleep until the monitor refreshed. According to a few point samples this reduced the power consumption of the mobile client application from around 30% of the total power usage to less than 3%.

While the processor is sleeping the application does not consume any power. However, after implementing this we noticed that the application occasionally rendered fewer frames. There was no longer a stable frame rate. This issue was tracked down into the sleep mechanism itself. The sleep mechanism is not deterministic. For example, if an application asks to sleep for 5 milliseconds it might end up sleeping for 10 milliseconds. This appeared to happen more frequently on the Android application than on the server.

After running some tests on the Android client we noticed that the sleep function appeared to have higher accuracy if lower sleep times were entered. For example, if sleep was set to 1 millisecond it would normally sleep no longer than 2 milliseconds. But if 10 milliseconds were entered it would be normal for it to sleep for more than 15 milliseconds. This leads to the following algorithm for sleeping presented in a C-like pseudo code language, presented in code listing 3.1.

```
target_sleep_time = get_target_sleep_time();
sleep_interval = 1000_000; // 1 millisecond or 1000_000 nanoseconds
sleep_stop = 1500_000;     // 1.5 millisecond
while(target_sleep_time - get_current_time() > sleep_stop)
{
      sleep(sleep_interval);
}

wait_for_vsync();
```

Code listing 3.1: The sleeping algorithm in pseudo code.

This algorithm sleeps in steps of `sleep_interval`. When the time slept has reached a point where it has less than `sleep_stop` time left to sleep, it falls back to the old mechanism of waiting for the screen to be synchronized. This has led to a more stable frame rate while still keeping power consumption low. This algorithm is used on both the server and the client. Figure 3.1 attempts to visualize the execution of the game loop.

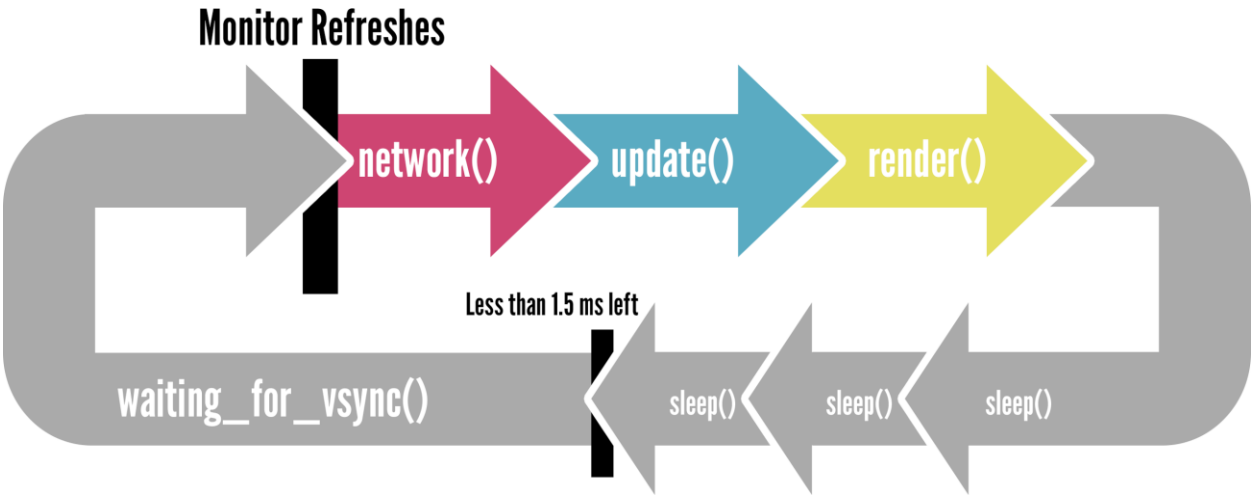

Figure 3.1: The game loop.

Server

Given the powerful computers we have today, reaching the targeted frame rate was not difficult. We still performed some basic optimization by batching rendering commands and avoiding garbage collection cycles.

The garbage collector design utilized in D employs a stop-the-world strategy for garbage collection [33]. This means that all threads pause and wait for the garbage collection cycle to finish every time a garbage collection is run. This becomes a problem when a fixed target deadline has to be reached. The duration of such a garbage collection cycle depends on the amount of memory currently used by the application. Even in a small game a garbage collection cycle can span dozens or hundreds of milliseconds. This would drop several frames during a collection. Several successive frame drops is a performance worst-case scenario and hence allocations in the main game loop were disallowed. All needed memory resources were allocated before the game enters its main loop.

Graphics cards work much faster if they are processing items in batches [34]. Instead of sending a single image to be drawn by the graphics card, thousands of times each frame, it is better to send many images to be drawn at the same time a few times per frame. This is called batching and the implementation goes through great lengths to make graphics batches as large as possible. For example, texture atlases are used to be able to render many different objects at the same time [35].

Client
Our client games are relatively simple in nature; because of this reaching the targeted frame rate was painless. It follows the same rule as the server concerning batching of rendering commands but when it comes to garbage generation the client is more liberal with allocations.

The garbage collector in Lua is very powerful for game creation, since it allows for iterative garbage collection [36]. The Lua garbage collector does not need to run for a whole cycle at a time and can instead be run for a short time each frame. The clients way of dealing with this is that the Lua garbage collector is given a few milliseconds each frame to handle any potential garbage that was created during that frame. In the games created this has worked well.

4. Platform
This chapter describes the core of the platform, starting with what is perhaps the most critical part - the network. It also introduces the Simple Data Language, used for serialization, and outlines the game related technologies concerning graphics and sound. The chapter ends with a description of the automated content reloading system utilized by the project.

4.1 Network
One of the most, if not the most important part of the platform, is the network implementation. This section will discuss the various methods and problems associated with implementing a local client-server model, based on synchronous and asynchronous usage of the transmission control protocol (TCP). The section also demonstrates the custom connection management protocol and messaging protocol used. Additionally, how different game specific messages are synchronized across the phone and the computer is discussed. This is followed by a subsection detailing how files are sent and synchronized across the different devices. Finally, this section is made complete by an analysis of what kind of latency can be expected from the system and how this relates to what kinds of games are appropriate for the platform.

4.1.1 Client-server model
A goal of the project is for the computer to emulate a video game console and the smartphones to emulate game controllers. The controllers of normal video game consoles do not communicate with each other. This applies to our system as well, and leads to a classic client-server network model. The computer (the console) acts as the server and the smartphones (the controllers) act as clients. Given this there is no direct communication between the phones. Any possible phone to phone communication has to go via the computer. In the network section, the word client refers

to a smartphone connected to the local network and server refers to the computer in the local network which the clients connect to.

4.1.2 Bluetooth versus Wi-Fi

The computer and the phones need to be connected wirelessly in order for gameplay to be practical. There are two major competing standards for consumer grade wireless communication, Wi-Fi and Bluetooth. Both of them have their distinct advantages and disadvantages which we present below.

1. While Bluetooth is available in most laptops, it is not available in all desktop computers. Even if a desktop computer does not support Wi-Fi, it is probably connected to a router in the local wireless network via an Ethernet cable, and can thus be connected to. Therefore, Wi-Fi is more prevalent than Bluetooth.
2. There is no widely used cross-platform Bluetooth library for the desktop, and the D programming language has no existing Bluetooth bindings. This makes Bluetooth impractical for our implementation of the server.
3. Bluetooth uses different protocols than the ones given by the Internet Standard (the protocols in use on Wi-Fi). These protocols are less documented and harder to find tutorials for.
4. Bluetooth handles server discovery through the somewhat cumbersome Bluetooth pairing process. Such a process does not exist on Wi-Fi and has to be implemented by the programmer herself.
5. Bluetooth consumes less power [37].
6. Most users have connected to a Wi-Fi network at some point. Bluetooth uses a completely different process, which might be foreign to users.

After evaluating the properties of the two technologies, we came to the conclusion that Wi-Fi was a better fit for our purposes.

4.1.3 Protocol

When deciding what underlying protocol to use, we employed the following criteria:

1. The protocol must be available in the implementation languages on the phones and the computer.
2. The protocol should have low overhead to minimize the strain on the local network (allowing many players to play at once).
3. The protocol should allow for reliable, in-order message sending.
4. It is preferable if the protocol also allows for unreliable messages.

If possible, it should not require great effort on our behalf to get it running.

The fourth criterion above needs some additional explanation. Why would one want unreliable messages in the first place? Unreliable messages put a lower strain on the network than reliable, in-order messages [38]. This can be very useful when speed of delivery is more important than ordered reliability. If a message is received out of order in a reliable, ordered system, the system

must wait for an older message to arrive before propagating the arrived message to application logic. An example where ordered reliability does more harm than good is the process of updating phone sensor data. The server wants as fresh data as possible, and if a newer message with sensor data arrives before an older message, the server wants to use that data right away. This is not possible in a reliable, ordered system.

Given this the options we considered were TCP, and a reliable version of the user datagram protocol (UDP). TCP has the benefit that it requires no effort to get reliable, in-order messages. It is available on all operating systems and in all the languages used. However, it does not support unreliable message sending. Reliable UDP is not available in all languages considered. Therefore, if we wanted reliability on top of UDP we would have to build this ourselves. This functionality would have to be implemented in each language responsible for the network layer. In the end we valued ease of implementation more than the possibility of removing latency, and the TCP protocol was chosen.

The server uses asynchronous TCP for all incoming messages and for most outgoing messages (see 4.1.8 for the only exception). This is done through the use of non-blocking sockets, with the intention of removing the need for special purpose networking threads. The network code instead runs directly on the main thread, thus avoiding the complexity of having to manage communication between several threads.

The asynchronous nature of non-blocking sockets leads to some problems related to messaging. At any time data is read or written from a socket, it is possible that the read/written data does not form a whole message. If this is the case the server remembers that partial message data and resends it at a later point in time.

4.1.4 Server Discovery

For any communication to take place, the clients need a way to find a server to connect to. It is preferable if this process is automatic so that the users need not be concerned with low level details such as the internet protocol (IP) address of the server or the port it is running on.

The server discovery process should ideally work on a local level. It should be possible to find multiple servers at once and the solution should work for phones using the Android operating system. Given these requirements many possible techniques, like having a known remote discovery server, were discarded.

The techniques to choose from in the end were UDP broadcasting and UDP multicasting. Of these we choose to use UDP broadcasting instead of multicasting since we did not need the additional feature offered by multicasting (the possibility to send messages to remote networks [38]). Multicasting is also a more complex technology than broadcasting, providing additional incentive to use broadcasting.

UDP broadcasting

UDP broadcasting works by sending a normal UDP-packet to a network specific broadcast address. This type of broadcast is called a directed broadcast. To get this broadcast address both the IP address and the subnet network mask need to be known. With them the subnet specific broadcast address can be calculated. This is done using the following formula [39]:

```
Broadcast address = <IP address> OR (NOT network mask)
```

Example: In a network with a server that has the IP 192.168.2.15 and a subnet network mask of 255.255.255.0 the broadcast address will be 192.168.2.15 OR (NOT 255.255.255.0) = 192.168.2.255

The implementation does not have access to the subnet network mask so it makes the assumption that the network mask is 255.255.255.0. This is the standard network mask for home networks / phone hotspots [40]. This limits the server discovery implementation process to only work on simple networks which do not employ customized network masks.

Server discovery protocol
By using UDP broadcasting, it is possible to communicate with all devices in the network. This is used by the server, which continuously sends UDP messages (containing all the necessary information needed to connect), on a specific port to all devices in the network. For the clients to receive a message they need to listen on the port the server is sending messages to. Therefore, a port must be agreed on beforehand. Our implementation uses port 7331 for this purpose. Table 4.1 shows the structure of this type of message, using an example message.

	Tag	**IP**	**Port**	**Server name length**	**Server name**
Example	"PPS"	192.168.1.25	9572	8	"PontusPC"
Type	3 byte ASCII string	uint32	uint16	uint16	utf-8 string

Table 4.1: A typical server discovery message. The second row shows example values. The third row shows the types of the different parts of the message.

PPS stands for **Project Party Server** (Project Party was the working name of the project) and is used to make sure that a UDP message received by a client is actually from a computer running the server application and not from some other device in the network. If a client receives a message that is not tagged with **PPS** it ignores that message. The IP address and port is all the information needed to connect to the server. The name is used as a human readable identification of the server, to tell different servers apart. It is used by the client when displaying what servers are available. The implementation uses the name of the computer as the server name.

4.1.5 Connection
When a server has been found, a client can proceed to make a connection. The connection is made in a series of steps:

1. A client makes a TCP connection attempt to the server.
2. The server accepts the incoming connection.
3. The server gives the incoming connection pending status.
4. The server generates a unique session ID and associates it with the incoming connection.

5. The server sends the session ID to the client.
6. The client receives the session ID.

At this stage is it possible that the connection is a reconnection. A client has lost connection and needs to reconnect. In the following list, items labeled by “a” specify a connection attempt, and the items labeled by “b” specifies a reconnection attempt.

7.a The client sends the received session ID to the server.
8.a The server receives the session ID.
9.a The server changes the connections status to active from pending.
10.a The connection process is done.

7.b The client discards the session ID.
8.b The client sends an old session ID to the server.
9.b The server receives the old session ID.
10.b The server checks if it remembers the old session ID from a previous connection.

Depending on whether the server remembers the old session ID or not, the server needs to accept or reject the incoming connection. These different scenarios are described with the “c” (accept) and “d” (reject) subscripts.

11.c The server has used the old session ID, the server changes the connection status to active from pending.
12.c A message is sent from the server specifying that the reconnect was accepted.
13.c The reconnection process is done.

11.d The server has not used the old session ID.
12.d The server sends a message to the client that the reconnect was not accepted.
13.d The server forcefully closes the connection.

Figure 4.1 shows the connection protocol in a flow chart form. As can be seen there are three different ways that the protocol can terminate. These are normal connection, successful reconnection and failed reconnection.

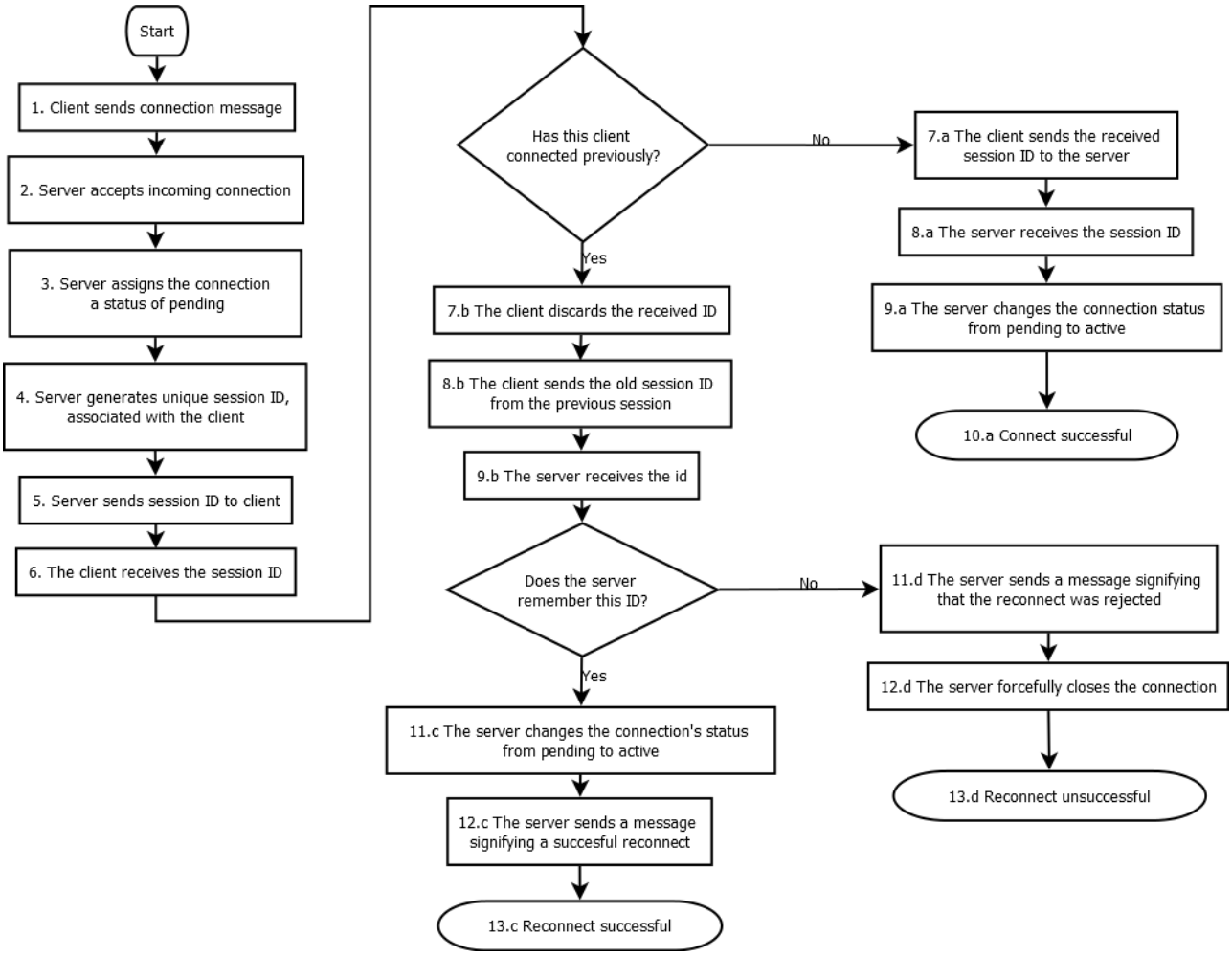

Figure 4.1: The connection process.

User interaction

From the user's perspective only a few steps are required in order to connect to a server. Once the application is started the screen in figure 4.2 is shown. From this screen the user may enter her desired alias. The application searches for active game servers automatically upon being launched. If no server is active when the application is started the list of active servers can be found by tapping the “Refresh” button. The user connects to a server by tapping the name of the computer hosting the desired server in the list of active servers displayed below the refresh button. It is required that each player is connected to the same network as the computer hosting the game.

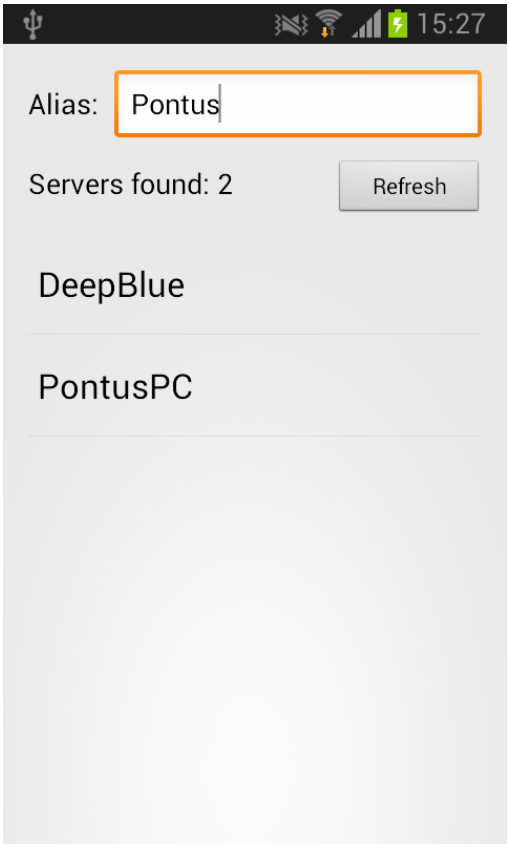

Figure 4.2: Server discovery screen. The user enters her desired alias into the textbox, searches for servers by tapping the refresh button, or connects to a server by tapping a server name (e.g. DeepBlue).

4.1.6 Disconnection

This subsection describes the different ways a connection might be dropped and what action is performed in each case.

Client side

Clients will disconnect from the server under any of the following circumstances.

1. The Android activity enters an idle state.
2. An invalid message is received.
3. The phone receives a special shutdown message from the server.
4. The phone exits the game.
5. Timeout. A message has not arrived from the server for 30 seconds.

If a phone disconnects due to the activity entering an idle state (1), it will reconnect again when the activity once again enters an active state. If an invalid message is received (2) the phone preemptively exits the game. If the disconnect is caused by the shutdown message (3), the phone will exit the game. In the event of a normal exit (4) the phone simply closes the connection via standard TCP shutdown before the game exits. Lastly, if the server times out (5) what to do is left for the specific game to decide. This usually happens if a server was shut down incorrectly or if

the phone got disconnected from the local network. In this case it is not clear how the client should react, which is therefore left to the specific game to decide.

Server side
A connection will disconnect from the server in the following circumstances.

1. The server is shutdown.
2. An invalid message is received.
3. Remote TCP shutdown request.
4. Timeout. A message has not arrived over the connection for 30 seconds

If the disconnection happens due to the server being shut down (2) the server will close the connections via a special shutdown message. If a message which does not conform to the message specification given in the next section is received by the server (2) it is assumed that the remote client is corrupt and the connection is terminated via TCP shutdown. The connection will be closed if the remote client requests connection termination via TCP shutdown (3). If a timeout occurs (4) the connection is closed via standard TCP shutdown.

4.1.7 Messages
This subsection gives an in depth description of how messages are handled by the networking system. It begins by defining the message specification, followed by describing how new messages are created. Lastly a discussion about how messages are kept in sync over the different languages and the initial problems of our approach is presented.

When a client has connected to the server they can begin to communicate by using messages. We wanted a highly extendable message format with as little overhead as possible. From these requirements we developed the message specification as seen in table 4.2.

Length	**ID**	**Content**
uint16	uint8	array of untyped data, with a size of Length - 1

Table 4.2: Message specification.

“Length” signifies the length of the message in bytes including the ID but excluding the length field itself. This field allows the network layer to split the incoming binary stream into discrete messages. The type uint16 makes the maximum message size 64kb. The ID field is used to uniquely identify the type of message sent. This ID is tied to the actual content in the message and is used to determine how to serialize/deserialize the message. Table 4.3 shows an example of a message containing phone sensor data.

Length	ID	Accelerometer	Gyroscope
1 + 12 + 12 = 25	phoneSensorID == 1	[0.34, 0.52, 1.34]	[0.42, 1.23, 0.64]
uint16	uint8	float[3]	float[3]

Table 4.3: A phone sensor data message. float[3] denotes a construct containing three floating point numbers.

Given that the ID is only a byte in size the number of standard incoming messages/outgoing messages is fixed at 255. Messages are divided into two different categories, system messages that exist for all games and game specific messages. The system messages start at an ID of 0 and progressively use higher IDs. Game specific messages start at an ID of 50, which was decided in the early stages of the network implementation. This was done to allow for the number of system messages to grow without conflicting with the IDs of existing games. Currently, only 9 system messages have been created.

Little endian format

Contrary to most data sent over a network the messages are sent and processed in little endian format, instead of standard network order - big endian format [38]. This was initially done to be able to send raw unprocessed structs from the server to the clients. But due to the fact that Android phones cannot perform unaligned operations, this had to be abandoned. This was discovered fairly late in the network implementation process (when we attempted to read messages on an unaligned boundary) and raw struct sending was discarded in favor of a more powerful automatic reflection based process (see next section). After the message sending routine had been changed there was little motivation of redesigning the system to use the big endian format, so the little endian format stayed.

Automatic message serialization through reflection

Using the D language's powerful compile time reflection capabilities [21] it becomes possible to automatically serialize and deserialize network messages. Annotated structs are used to specify messages. Code listing 4.1 gives an example of how one would specify an incoming network message and an outgoing one.

```
@IncomingNetworkMessage(someID)
struct MyIncomingStructMessage {
   int aNumber;
   char[] aString;
}

@OutgoingNetworkMessage(someOtherID)
struct MyOutgoingStructMessage   {
   int aNumber;
   double anotherNumber;
}
```

Code listing 4.1: An outgoing network message struct and an incoming message struct.

The name that follows the '@' symbol is a D style annotation. It gives additional information that is needed by the system in order for it to handle the message. If it is of type ***OutgoingNetworkMessage,*** the serialization code knows that this message will be sent from the server to the clients. If it, on the other hand, is of type ***IncomingNetworkMessage,*** then it knows this message will be received by the server from a client. The *someID/someOtherID* constants represent the ID which is part of the message header, previously mentioned in this section. Table 4.4 gives an example of a serialized instance of the struct MyOutgoingStructMessage.

Length	ID	aNumber	anotherNumber
1 + 4 + 8 == 13	someOtherID	5	563.51266
uint16	uint8	int32	double

Table 4.4: Message generated from the struct MyOutgoingStructMessage.

Message serialization synchronization across language borders

The messages sent over the network are extremely sensitive to changes in the message data section. The data has to be read in order and the type of each individual field has to be considered. When a change is made it must be made on both the server and the client. If a change is only made to one and not the other, the system is likely to crash or, even worse, silently do the wrong thing.

Initially the server and the client messaging code were kept in sync by hand, a rather tedious process. If a change was made, one had to remember to change code in several different places, in different languages. This required immense discipline and was not a task one would perform without good reason. In order to remedy this situation, an automatic system, which kept the serialization of messages synchronized across languages, was implemented.

The aforementioned system generates the required Lua serialization network code from the struct message specifications mentioned previously in this section. Since the Lua code is generated from the D code it will always be in sync. To make sure that the client always has the most recent version of the Lua network code, it is sent to the client after each successful connection attempt.

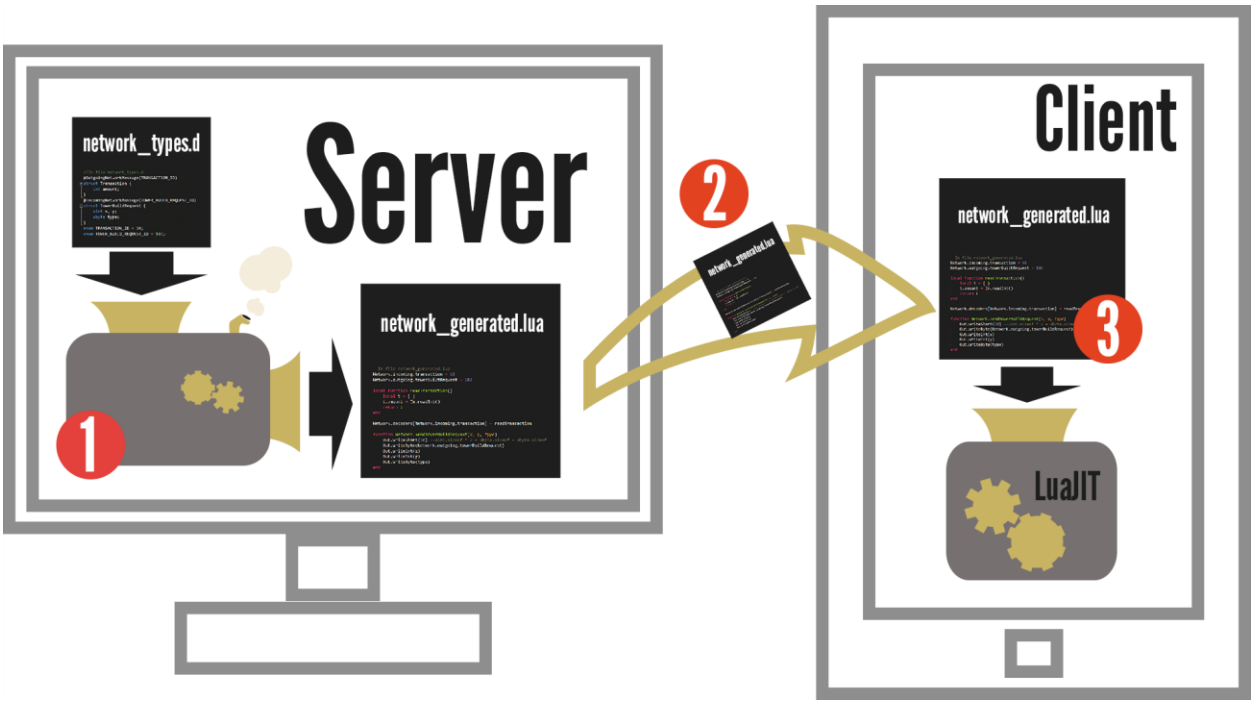

Figure 4.3: Generation and synchronization of network serialization code.

Figure 4.3 shows this process in action, with the distinct stages marked with numbers. The process begins by generating the required Lua serialization network code from the struct messages specifications (1), and then it sends the generated Lua source code over the network to the client (2), which feeds the source to the LuaJIT interpreter (3).

Code listing 4.2 gives an example of the generating code, and code listing 4.3 displays the generated code for two simple network messages.

```
//D code (Generating code)
@OutgoingNetworkMessage(TRANSACTION_ID)
struct Transaction {
    int amount;
}

@IncomingNetworkMessage(TOWER_BUILD_REQUEST_ID)
struct TowerBuildRequest {
       uint x, y;
       ubyte type;
}
enum TRANSACTION_ID = 50;
enum TOWER_BUILD_REQUEST_ID = 102;
```

Code listing 4.2: An outgoing transaction message struct and an incoming tower build message struct.

```
--Lua code
Network.incoming.transaction = 50
Network.outgoing.towerBuiltRequest = 102

local function readTransaction()
    local t = { }
    t.amount = In.readInt()
    return t
end
Network.decoders[Network.incoming.transaction] = readTransaction
function sendTowerBuildRequest(x, y, type)
    Out.writeShort(10)
    Out.writeByte(Network.outgoing.towerBuildRequest)
    Out.writeInt(x)
    Out.writeInt(y)
    Out.writeByte(type)
end
```

Code listing 4.3: Generated serialization/deserialization Lua code for the network messages in code listing 4.2.

The interesting pieces of the generated Lua code are the `readTransaction` and `sendTowerBuildRequest` functions. The first function reads a transaction message, puts that message in a Lua table and returns that table. Lua code that is interested in transaction messages will get notified when a transaction message arrives and can use the created Lua table. The second function takes all necessary input from its parameters and serializes them to the network stream. *In* and *Out* are Lua tables that enable code to either read from the network stream or write to the network stream.

Having a message communication system based upon generated code has some nice properties. First, it is simple to add new messages, only add a new struct to the system. Secondly, if the message format should change some time in the future, the only thing that has to change is the code generation code. Lastly code generators can easily be adapted to emit logging code or extra type checks which can make finding a bug much easier.

4.1.8 Content transmission

At an early stage of the project we decided that we wanted a framework where it was possible to create game clients entirely on the server side. For this to work, we needed a way to send everything the client makes use of from the server to the phones. This included things such as images, fonts, and Lua game logic code. However, the messaging scheme normally used did not allow for messages larger than 64kb. This was problematic, since many of our assets were larger than 64kb. The problem was solved by extending our standard message format, thus making file transfer a special operation.

Files are sent in a three step process. First a file transfer message header is sent through the standard message sending routine. Secondly the entire file is streamed directly from the server's hard drive to the client's internal memory. Thirdly, the received file is saved on the external

storage device (SD-card) on the client. Given that the file is stored on an external storage device phones are required to have an SD-card or similar external storage device for the application to function. The file transfer message is given in table 4.5 below.

Length	ID	Relative path length	Relative path	File size
1+2+32+8=43	2	32	“achtung/phone/textures/pixel.png”	165
uint16	uint8	uint16	ASCII string	uint64

Table 4.5: File transfer message.

The relative path is simply the path to the file, relative to the server’s resource directory. The file size is the size of the file to be sent. A path relative to game_name/phone is used when loading the file from Lua game scripts, where game_name is the name of the game running.

Keeping files in sync

In order to always keep the smartphone resources (e.g. textures, scripts) up-to-date, all resources are kept on the server side. All phone related assets of a game are sent to the client from the server upon every successful connection and reconnection. The approach of sending every file upon connection has an obvious shortcoming. Even if the player already has all the files on her phone, she has to wait for all resources to be sent over the network, and saved in physical memory. This was done in order to make sure that the client and the server always run the same version of the game. As the process of sending the assets from the server to the clients is completed in a matter of seconds, an overhaul of this system has not been prioritized. This small downtime could be sidestepped with a simple system of version control. We will describe such a system in section 6.4.

4.1.9 Latency

In this subsection, we discuss the issue of input latency. Included are two latency tests comparing different network parameters. The first test examines how latency is affected by turning on the socket option TCP_NO_DELAY, while the latter compares the latency of the UDP and the TCP protocols.

At the start of the project, the games suffered from an unacceptable amount of latency. After some investigation we discovered that the cause of this might have been Nagle’s algorithm [41]. TCP optimizes for bandwidth by sending fewer packages at the expense of higher latency. This is done using Nagle’s algorithm. This algorithm prevents messages from being sent immediately, and instead buffers small messages to send them in batches. This is problematic in soft real-time applications such as games which require low response time. Fortunately, all TCP implementations used provide an option to disable Nagle’s algorithm. This is done by setting the socket option *TCP_NODELAY*.

By disabling Nagle's algorithm, latency was lowered to the point of no longer being a major issue in our games. We tested the latency of TCP with Nagle's algorithm enabled, and disabled using a modified version of the server of the Achtung game, which would record how many milliseconds passed between the packets it received. The tests were carried out on two different mobile hotspots. In order to realistically simulate the conditions of gameplay, three smartphones were connected to the server. The results of this testing are shown in figure 4.4. The x axis represents the packet number of the arrived packet, and the y axis represents the time interval between the reception of that packet and the packet preceding it. As can be inferred from the figure, Nagle's algorithm periodically causes latencies above 2000 ms, which is an unacceptable level in all games the platform was made for. In conclusion, we recommend disabling Nagle's algorithm when latency is an important factor.

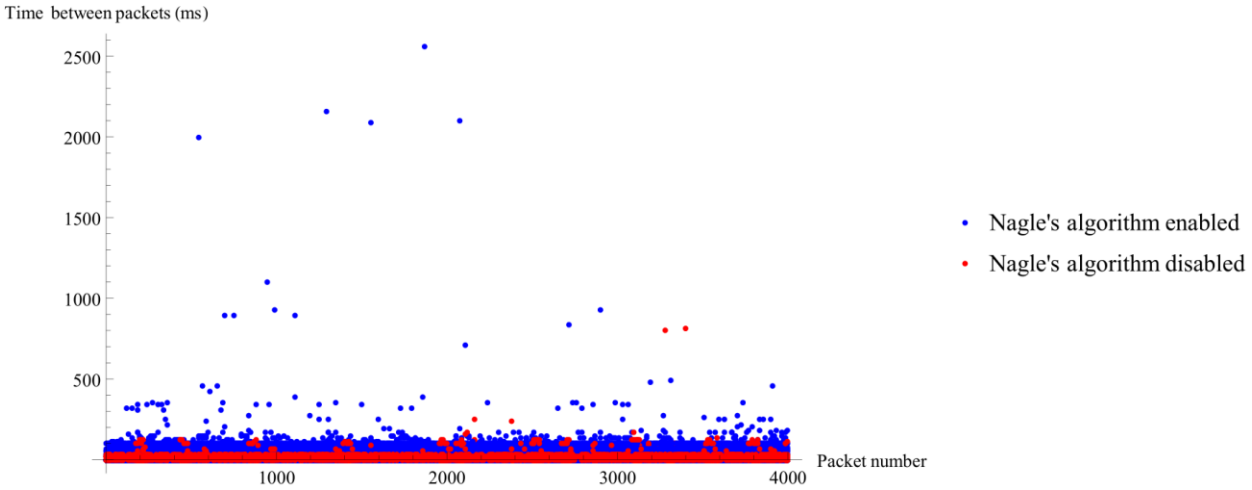

Figure 4.4: Comparison of TCP latencies with Nagle's algorithm enabled/disabled.

As previously mentioned, TCP is the main network protocol used by our platform. However, we wanted to examine whether UDP was a better choice for sending sensor data. In order to test this claim, we created a basic UDP implementation with the sole purpose of sending sensor data.

We compared TCP to UDP using the modified Achtung client previously mentioned, using two mobile hotspots. Once again, three phones were connected to the server in order to simulate gameplay. The results are shown in figure 4.5. While comparing TCP to UDP, we discovered that TCP yielded local latency maxima in the vicinity of 800 ms (visible as the two clear outliers of figure 4.5) When the test was repeated using UDP no observation ever exceeded 250 ms. It has been shown that players on a LAN network were able to tell the difference in gameplay when latencies were as low as 75 ms in the game Unreal Tournament 2003 [42]. If we base our criteria of success on this research, the system could benefit from additional latency reductions. Both transmission protocols produced suboptimal results that might impact the users experience while playing a game. However, UDP was found to be the better option in terms of latency spikes.

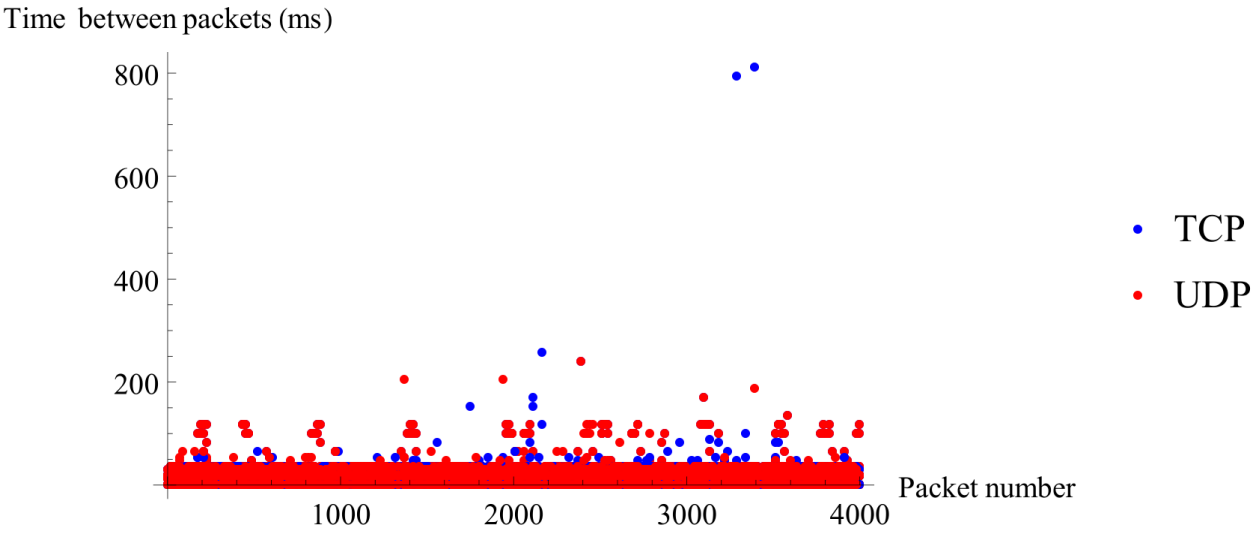

Figure 4.5: Comparison of TCP and UDP latencies.

Some games are more tolerant of higher latencies. Traditional turn-based games such as Chess are obviously not affected by the latencies we observed in our system, while action oriented games similar to the previously mentioned Unreal Tournament 2003 might be impacted negatively. In our personal experience playing the games produced for our platform, latency usually is not a factor. This might be explained by the above experiments. Out of the packets sent using TCP in the second experiment, over 99.64% of packets arrived in less than 75 ms, the point at which latency becomes noticeable. While this is encouraging, we would still recommend implementing games which are tolerant to some amount of latency (for example, our Pictionary game from section 5).

4.2 Serialization using the Simple Data Language

This section describes the Simple Data Language (SDL), used for serialization on the server side of the platform. It begins with describing why it was decided to create SDL followed by how it is used.

A good way to separate code from data is to serialize runtime objects into text documents using a human-readable data serialization format. Data formats such as the JavaScript Object Notation (JSON) [43] and the Extensible Markup Language (XML) [44] are widely used in the software industry, but left us unsatisfied, as they contain an excessive amount of redundancy. A format named Simple Data Language (SDL) was proposed and implemented to address these perceived issues of existing serialization languages. SDL was loosely based on the Simplified JavaScript Object Notation (SJSON) format of the Bitsquid game engine [45]. In table 4.6, one can see an example object being represented in SDL, SJSON, and JSON side by side.

layout.sdl:

```
// SDL

// Vertical bars
// are redundant
titleFont = Segoe54

// Yellow (ABGR)
titleColor = 0xFF00FFFF

initialRenderSize = 10_000

resources =
[
  {
    type = texture
    path = tex\pixel.png
  }
  {
    type = font
    path = |fnt\Segoe54.fnt|
  }
]
```

layout.sjson:

```
// SJSON

titleFont = "Segoe54"

// Yellow (ABGR)
titleColor = 4278255615

initialRenderSize = 10000

resources =
[
  {
    //Enum AssetType.texture == 0
    type = 0,
    path = "tex\\pixel.png"
  }
  {
    //Enum AssetType.texture == 1
    type = 1,
    path = "fnt\\Segoe54.fnt"
  }
]
```

layout.json:

```
{
  "_comment": "JSON",

  "titleFont": "Segoe54",
  "titleColor": 4278255615,
  "titleMargin": 50,

  “initialRenderSize”: 10000,

  "resources":
  [
    {
      "type": 0,
      "path": "tex\\pixel.png"
    },
    {
      "type": 1,
      "path": "fnt\\Segoe54.fnt"
    }
  ]
}
```

Table 4.6: Comparison between SDL, SJSON, and JSON by representing an example object in all languages.

The SJSON format is, as the name suggests, simplified JSON, and therefore differs from normal JSON in only a few aspects [45]:

- A SJSON file is assumed to always define an object. That is, there is an implicit pair of braces (“{ }”) surrounding the entire SJSON file.
- The equal sign = is used instead of the colon : to define object key-value pairs.
- Quotes around the key in key-value pairs are optional (unless the key contains spaces or the equal sign).
- Commas are optional in object and array definitions.
- C and C++ style comments are supported.

SJSON successfully remedies most of the redundancy found in the JSON language due to the changes mentioned above, but some additional modifications were still found to be desirable. There was no incentive to conform to the SJSON language, since SJSON is not an industry standard. The following changes were implemented on top of the SJSON format to create SDL.

- The double quotes were replaced by vertical bars (“|”), since this allows us to use double quotes inside of an SDL string, which is a much more common character inside of strings than the vertical bar. For instance, character dialogs often contain them.
- Double quotes for keys are explicitly disallowed.
- A string value need not be identified as a string by double vertical bars, as long as it does not contain spaces or escape characters.
- Commas are disallowed inside object and array definitions.
- Generic enums are supported.
- Comments that span multiple lines are not supported.
- Hexadecimal numbers are supported using 0x prefix.
- Underscores are ignored in numbers, allowing for numbers like 1_000_000 to be parsed.
- Escape sequences are not supported.

The features that were removed in the transition from SJSON to SDL were simply found to be unnecessary. Features such as escape sequences might seem a critical omission, but as previously stated they would not contribute in any meaningful way, and were not implemented as a result. We do recognize that this is an area of possible improvement, and leave it as a prospect for future work.

In order to let parsing of objects be as concise and generic as possible, a generic parser was created, capable of parsing generic structs of the D programming language. This was made easy by the excellent support for template metaprogramming of the D programming language.

4.2.1 SDL Usage

Idiomatic use of the parser is to create a new struct type which contains everything that should be parsed. An instance of the desired type can be instantiated by a single call to the method fromSDLFile. The benefits of this approach come from the automatically performed checks in the parser. These checks will print a descriptive error if a member of the object type was not found during parsing, (unless the member was annotated with the @Optional attribute, described later in this section) or if other errors occur. This approach aims to discover all errors in configuration files during the load sequence instead of generating garbage data. If a simple spelling error is made, a descriptive error is printed on load, instead of letting the corresponding object member retain its default value.

However, this is not the only way to use the parser. Instead of relying on an abundance of parsing integrity and runtime checks provided by checking against a type definition, a lazily evaluated iterator object can also be created. The iterator object does not contain any actual runtime data but instead parses objects as they are requested at runtime.

gameconfig.sdl:	types.d:	main.d:

maxPlayers = 4
 	
//WindowConfig
window =
{
width = 1280
height = 720
fullScreen = false
}
	struct WindowConfig
	{
	int width;
	int height;
	bool fullScreen;
	}
	
	struct GameConfig
	{
	WindowConfig window;
	int maxPlayers;
	}
		void main()
		{
		auto gameConfig =
		fromSDLFile!GameConfig(“gameconfig.sdl”);
		
		//This check passes
		assert(gameConfig.window.width == 1280);
		}

Table 4.7: Example code which parses the contents of gameconfig.sdl into an object of type GameConfig.

Table 4.7 gives an example usage of the parser. The fromSDLFile!GameConfig method parses the file gameconfig.sdl, puts the contents in a struct of type GameConfig. Subsequently, a runtime check is performed to make sure that the width of the window of the object is indeed 1280 as expected.

4.2.2 Annotation-based extensions

The D programming language permits users to define custom attributes, annotations to types, members and/or functions [46]. In order to extend the functionality of our parser, we utilized this feature by defining two custom attribute types, @Convert and @Optional. The aforementioned idiomatic use of the parser involves defining struct types in order to enforce parse-time error checks, but coupled with user-defined attributes these types can be imbued with additional information about their members.

Consider the struct type WindowConfig from table 4.7. Code listing 4.4 gives an example of this. We annotate the member *fullScreen* of type bool with @Optional(false), it tells the parser that the user is not required to specify a value for that member in the .sdl file, and that if no such value is found, the default value false will be assigned to fullScreen.

```
struct WindowConfig
{
     int width;
     int height;
     @Optional(false) bool fullScreen;
}
```

Code listing 4.4: A D struct highlighting the simplicity of using D annotation for specifying optional data.

Code listing 4.4 illustrates the power and simplicity of user-defined attributes. Ideally, users should not be forced to manually set all the fields of the struct type they are interested in parsing if one can conceive a commonly accepted default value.

The @Convert attribute has been found to be useful in situations where the user would prefer to specify the name of a file from which a member should be loaded, rather than having to specify the exact structure of the member in the same .sdl file.

Consider specifying a font in a layout object. Ideally, specifying the name of the font should be enough for the parser to retrieve the font ID of the named font. Instead of burdening the parser with new logic for every new type conversion, the converting function can be a compile time parameter to the @Convert attribute, which the parser will diligently make use of. An example is shown in code listing 4.5 which loads a font from a name.

types.d
```
auto stringToFont(string ID)
{
    return Game.content.loadFont(ID);
}
struct Layout
{
    @Convert!stringToFont() FontID titleFont;
    string title;
}
```

layout.sdl
```
titleFont = |helvetica_light72|
title     = |Achtung, die Kurve!|
```

main.d
```
void main()
{
    auto layout = fromSDLFile!Layout(“layout.sdl”);
    Game.renderer.addText(layout.titleFont, layout.font, float2(x,y));
}
```

Code listing 4.5: A D struct highlighting the simplicity of using D annotation for specifying optional data.

4.3 Graphics

This section gives a short overview of the graphical systems that were designed and used during the development of the platform. It begins by describing the core rendering functionality present on both the client application and the server application. This is followed by a description of the animation system and the particle system developed for our Tower Defense game from section 5.

The implementation of the renderers is based upon the OpenGL 3.3 graphics library on the server, and the OpenGL ES 2.0 graphics library on the phone. These frameworks are very similar to each other, which was helpful when porting the server renderer written in D to the client renderer written in C/C++. OpenGL (ES) was selected as our low level graphics API since it exists on all the platforms which we are interested in. It is also cross-platform; hence the graphics code currently running on the Windows operating system should work without modification if in the future it was ported to Linux or Mac OS X.

4.3.1 Images and text

The basic renderers that were implemented on the server and the client were inspired by the SpriteBatch renderer present in the popular Microsoft gaming framework, XNA [47]. Just like the SpriteBatch renderer the renderers created for our platform supports the rendering of items in batches (rendering several objects simultaneously). The batching is done to improve performance as discussed in the efficiency section 3.5. The items rendered are either transformed images or text.

To be able to render images or fonts these first have to be loaded from disc. The server uses the FreeImage library to load images, while the client uses LodePNG [48] to achieve the same. Fonts are created via an application called BMFont [49]. This application, which was created by Andreas Jönsson, allows users to convert regular TrueType fonts into bitmap fonts. These bitmap fonts can be loaded into the applications using custom content processing code, which converts them into a format usable by the different renderers.

Drawing images and rendering text is the only thing needed to create basic games. This is shown in all of the games we created with the exception of the Tower Defense game, which uses more advanced rendering technologies such as keyframe animations and particle effects.

Usage of the renderer is quite straightforward. Code listing 4.6 shows the complete process. A font and an image are loaded. After being loaded, the image and the font are rendered. The resulting image is shown in figure 4.6.

```
auto image      = Game.content.loadTexture("smiley");
auto smileFrame = Frame(image);
auto segoeUI    = Game.content.loadFont("Segoe54");

auto center = float2(Game.window.size / 2);
auto origin = smileFrame.dim / 2;
auto scale  = float2(1,1);
Game.renderer.addFrame(smileFrame, center,
                       Color.white, scale, origin);

auto text = "Drawing made easy!";
auto textSize = segoeUI.measure(text);
auto textPos  = float2(center.x - textSize.x / 2,
                       center.y - smileFrame.dim.y / 2);
Game.renderer.addText(segoeUI, text, textPos, Color.red);
```

Code listing 4.6: Code for loading and rendering the image in figure 4.6.

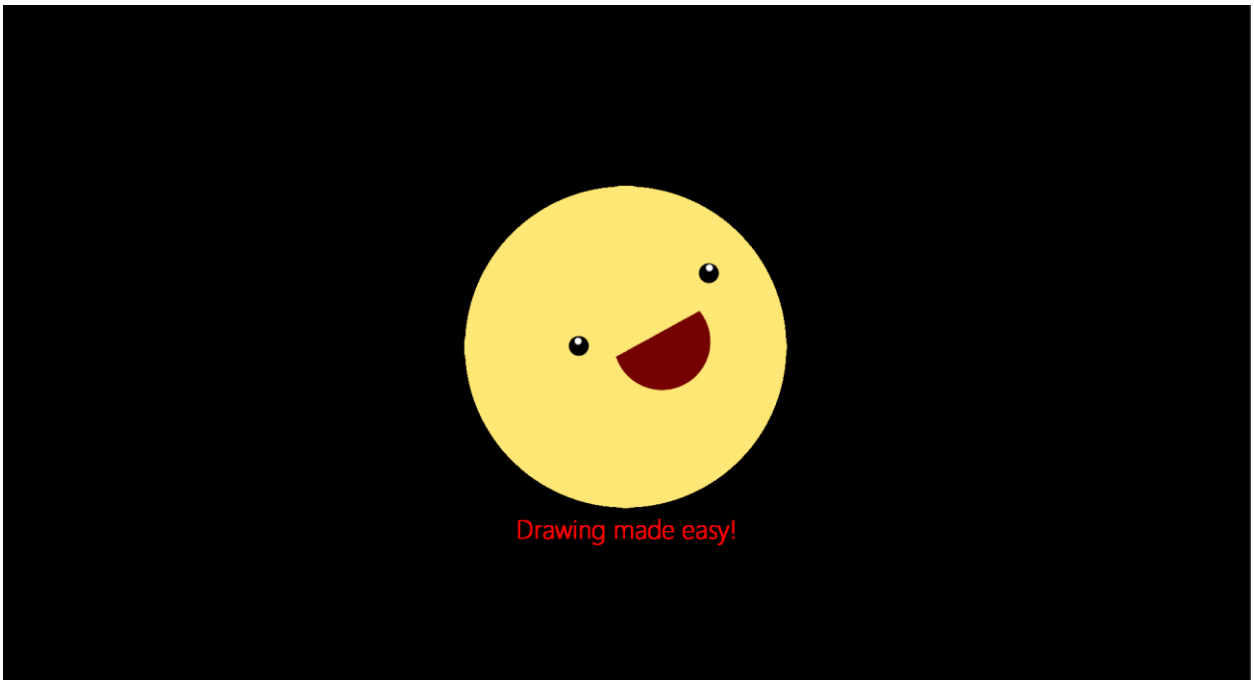

Figure 4.6: The rendered result of the code in code listing 4.6.

4.3.2 Animation system

During the development of the Tower Defense game, it was decided that to make the game more interesting it should feature more advanced graphical effects than simple moving images and text. This led us to integrate keyframed animations into the server component of the platform.

Animations were created via the free version of a 2D-animation editor called Spriter [50]. This editor allows a user to create custom keyframed object animations, which can later be played back in games. Figure 4.7 show the creation process of an enemy character inside the Spriter editor. Figure 4.8 shows the animation being used inside the tower defense game.

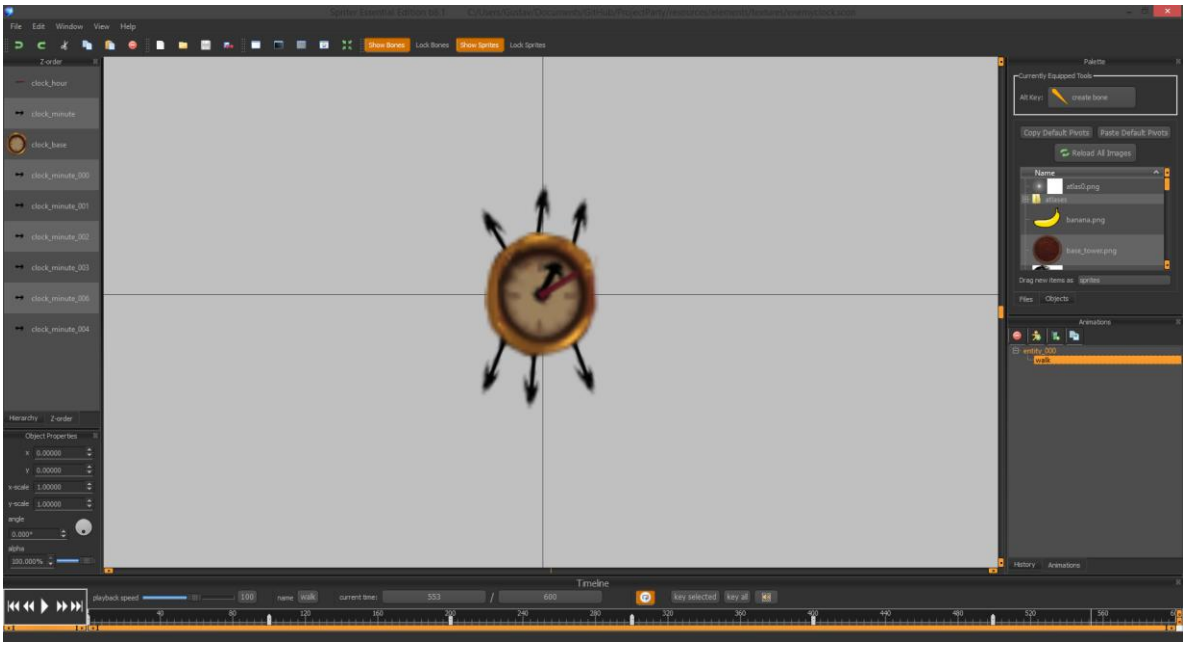

Figure 4.7: An enemy animation being edited in the Spriter animation editor

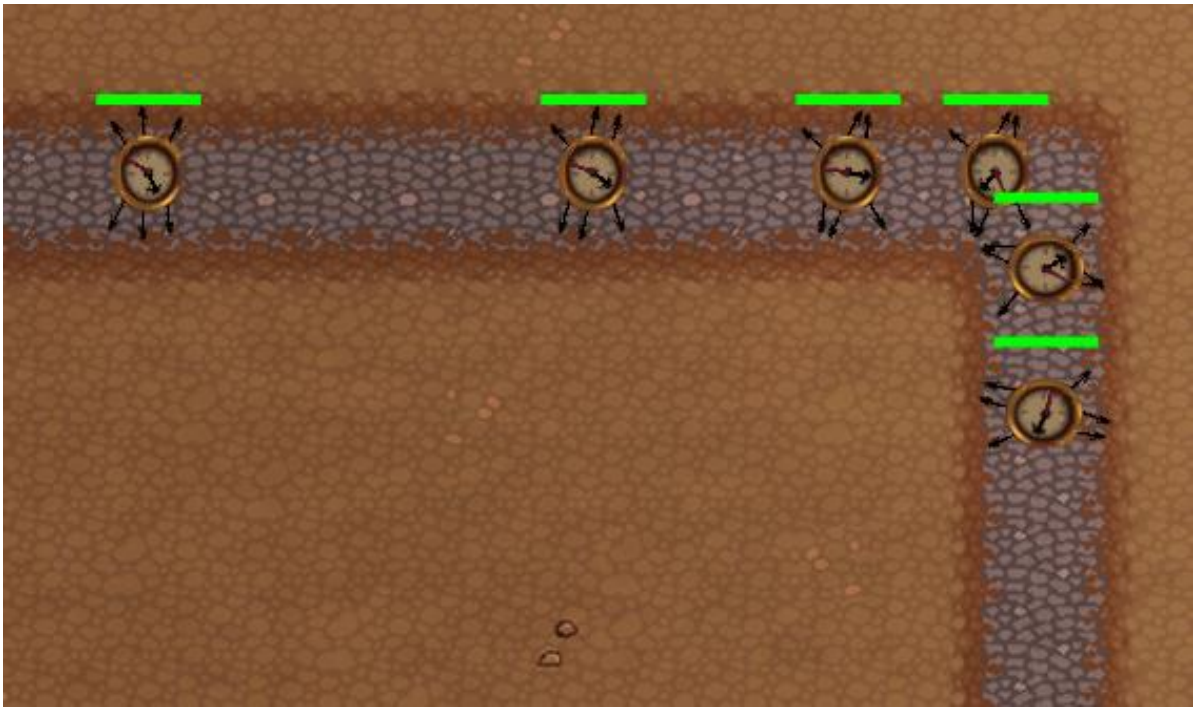

Figure 4.8: The enemy edited in figure 4.7 being rendered in the tower defense game.

4.3.3 Particle system

Particle systems are used to simulate dynamic entities such as smoke, water or explosions. Particle systems work by rendering many small images, together forming a group of particles which together emulate a dynamic entity. These entities are commonly referred to as particle effects.

The implemented particle system takes its inspiration from the Shuriken particle system present in the popular cross platform gaming framework Unity 3D [51]. Particle effects are composed by using one or more particle emitters. Theses emitters as their name implies are used to emit particles. What separates different emitters from each other is how they emit particles, and what type of particles they emit.

The particle effect system is highly driven by data. That is, many of its variables can be customized in order to achieve many different particle effects. These variables are specified in the SDL language discussed in section 4.2. Code listing 4.7 shows an example of a particle effect written in SDL. The example effect is the explosion effect used by rockets in the tower defense game. How this effect progresses as time goes by is shown in figure 4.9.

```
time = 0.32
playing = true
looping = false
emitters =
[
  {
    //Particle Common stuff
    startSize = { x = 100  y = 100 }
    endSize       = { x = 150  y = 150  }
    speed    = 150
    speedVariance = 100
    lifeTime  = 0.6
    startColor  = 0xaa0000FF
    endColor    = 0xaa000000
    colorVariance = 0x00008800
    rotationSpeed = 0
    rotationVariance = 5

    startAlpha = 0.5
    endAlpha = 0
    points =
    [
      { time = 0  count = 100  particle = fire },
      { time = 0.2  count = 20   particle = particle-smoke },
      { time = 0.3  count = 20   particle = smoke }
    ]

    angle    = 1.57
    line     = 0
    width    = 6.28
  }
]
```

Code listing 4.7: The SDL definition of the effect rendered in figure 4.9.

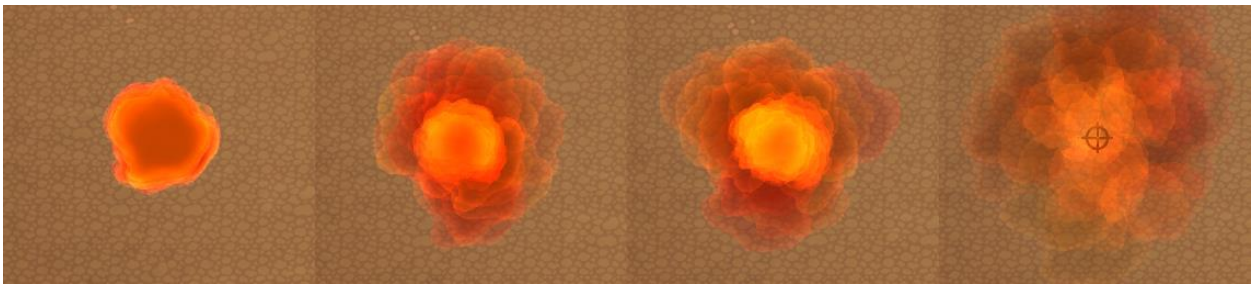

Figure 4.9: The explosion effect from code listing 4.7 being rendered by the particle system

4.4 Sound

This section describes how sounds add to the gaming experience and how sound is integrated in the platform.

Sounds play a vital part when it comes to the user's experience of any video game. A game contains many types of sounds, such as music which can be used to enhance the ambiance in a game or general sound effect such as explosions. The music is normally adapted to suit a specific part of a game; at nightly settings the music is often more soothing while the music in action oriented parts of a game is more energetic or even violent. Sound effects can be used to improve quality of a game by clarifying the result of a certain action. However, we have limited our sound implementation to only one type of music and a minimal number of sound effects.

The server implementation supports playing both music and any number of sounds effects through the SDL_mixer library. SDL_mixer stands for Simple DirectMedia Library mixer and has nothing to do with the Simple Data Language (SDL). The audio formats available are OGG, WAV, and AIFF. Sound loading has been integrated into the content reloading system described in the next section, making it possible to quickly integrate new sounds while testing the game.

Playing music or sound effects is not yet available on the phone. Plans have been made for including SDL_mixer on the phone as well.

4.5 Content reloading

This section describes content reloading, what it is and how we make use of it on the platform.

As soon as a resource has been changed in the file system, all copies of that resource in the game are out of sync. In order to reflect changes of files on disc, it is commonly necessary to restart the game. To prevent a resource in the game from being outdated, one could instead reload resources, as soon as the file from which the resource was loaded changes. This can greatly speed up development since no time is needed to restart the game.

This has been implemented in our system. Consider the following example: An artist wants to change a texture to better fit the environment. The game scene containing the texture can be running concurrently with an image editor, with the texture modifications being reflected in the game in real time. In figure 4.9, GIMP is used to insert a large hole into the map background of our Tower Defense game.

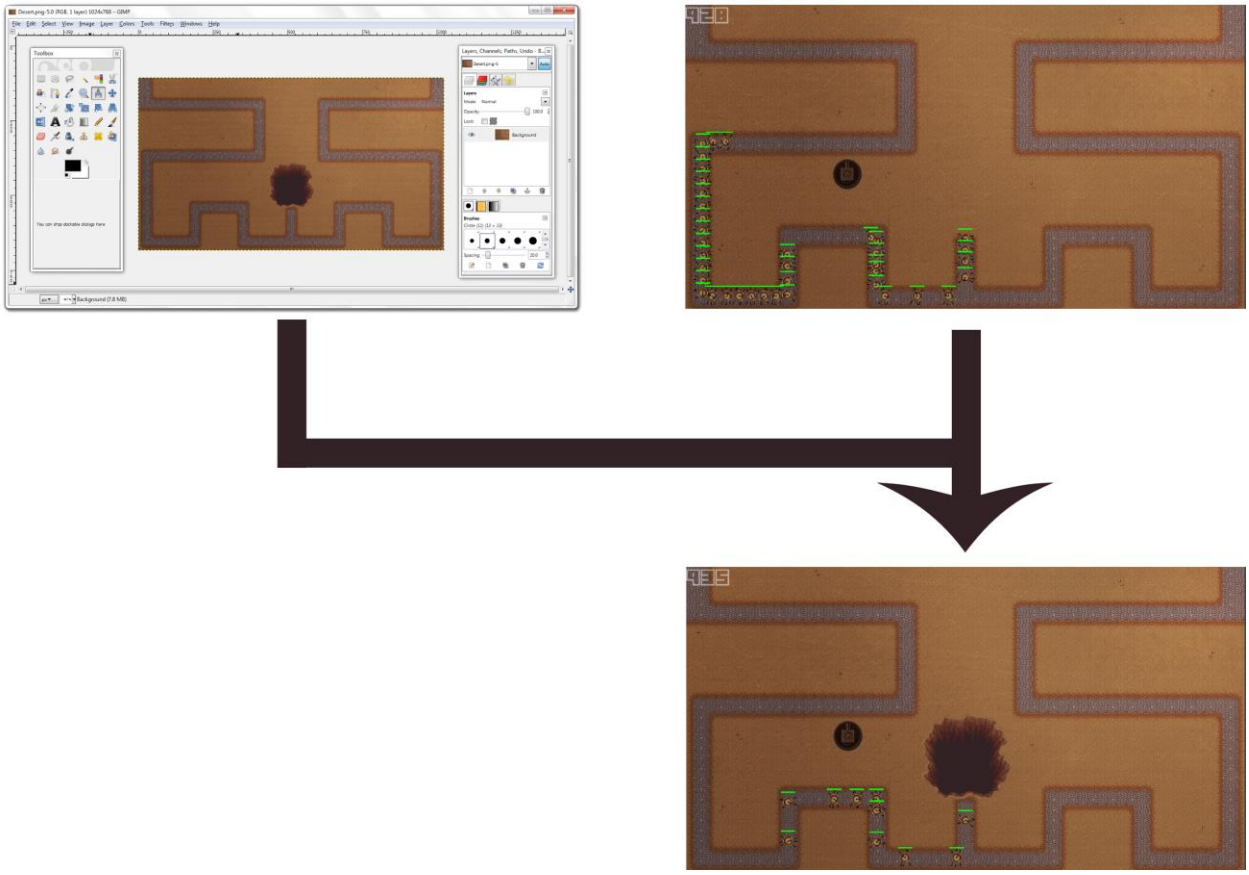

Figure 4.9: The image editor GIMP being used to edit the background of the Tower Defense game while the game itself is running.

Different operating systems expose different interfaces to their file systems. This is a problem when writing cross-platform software. This problem can be solved by defining a platform independent interface and using the conditional compilation features of the D programming language. However, only code making use of the Windows file system interface is currently implemented. Since runtime content reloading does not in itself contribute to any end product, and is strictly a tool to speed up the iteration cycles of the development phase, it might be acceptable to leave this feature disabled on other platforms.

5. Games

This chapter contains descriptions of the four games we developed to test the viability of the platform. Each section gives a motivation as to why we made that particular game and then we show how the games work and what they look like.

5.1 Achtung

Achtung was the first game created for the platform. It served both to test the general effectiveness of the platform and in particular to test if sending soft real-time controller input was possible.

Achtung, die Kurve is a competitive game made for two or more players where the objective is to be the last player left on the playing field. Each player spawns as a dot at a random point on the screen. This dot moves at a constant speed and all players can control in which direction their dots will move. As the dot moves, it leaves behind a line that becomes an obstacle. Colliding with this line results in a game over for the player. Colliding with the outer edges of the playing field also results in a game over. As the line continues to grow small openings are created at random intervals, which can be used to escape a certain game over. To utilize these are what distinguishes skilled players from the average ones. Figure 5.1 shows a gameplay session between two players. The blue player is currently in the lead with a score of three, against the other player's score of zero.

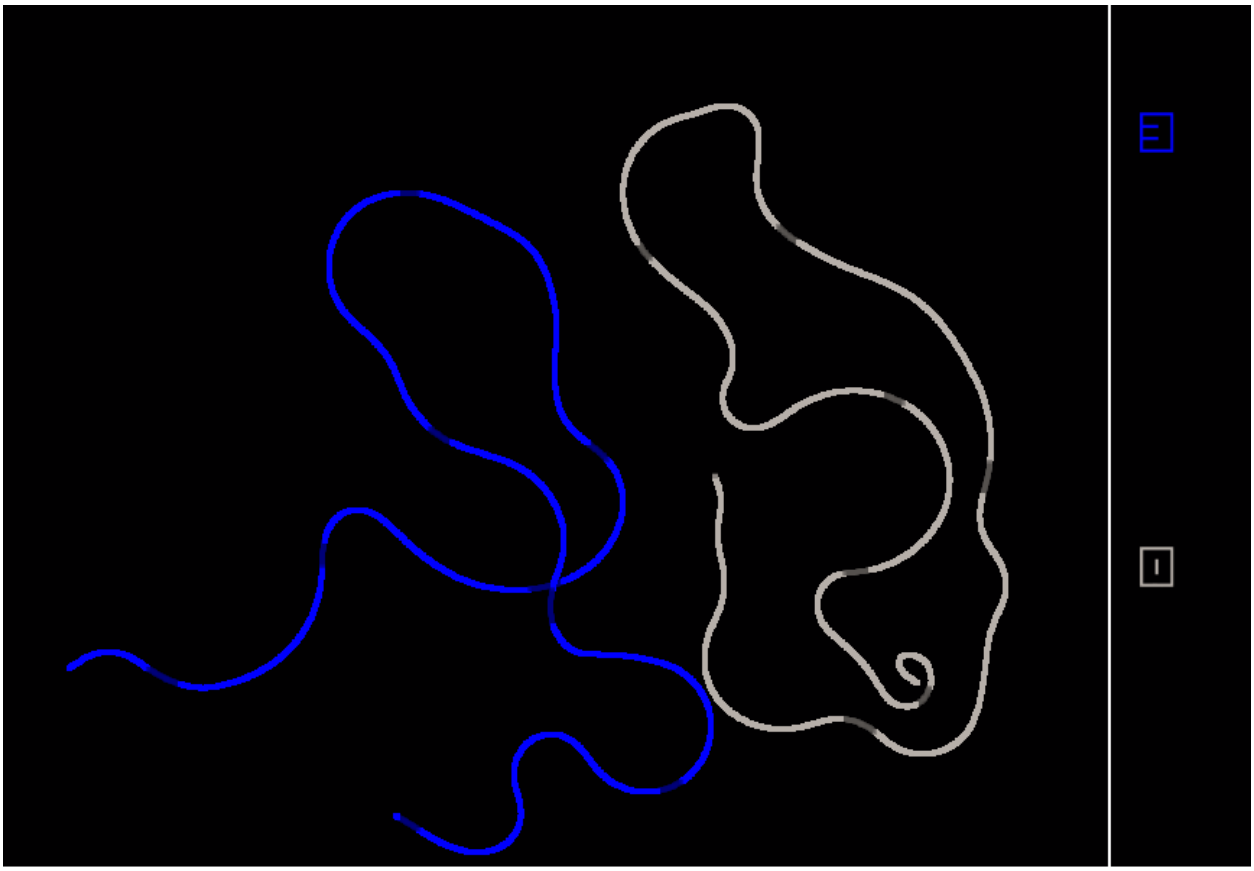

Figure 5.1: Achtung being played by two players, the numbers to the right of the playing field being the players' scores.

To control the direction in which the worm is moving the player tilts her phone. In this way, the phone's built-in accelerometer is utilized for steering.

5.2 Tower Defense game

A cooperative game in the Tower Defense genre was also developed. In a Tower Defense game, the goal is to prevent computer controlled enemies from reaching their goal, by building towers which kill the enemies by launching projectiles. This version comes with a twist, as the players are required to control the towers manually in order to make them fire, instead of towers working autonomously as is usually the case.

The Tower Defense game is played on a grid based map that is divided into buildable territory and the path that the enemies are following. The objective is to prevent waves of enemies walking down a predetermined path from reaching the end of it. This is done by building defensive structures that shoot down the enemies. Each wave consists of more powerful enemies than the previous one and more potent towers are needed to prevent them from reaching their goal. The game is over when a certain number of enemies have reached their goal at the end of the path.

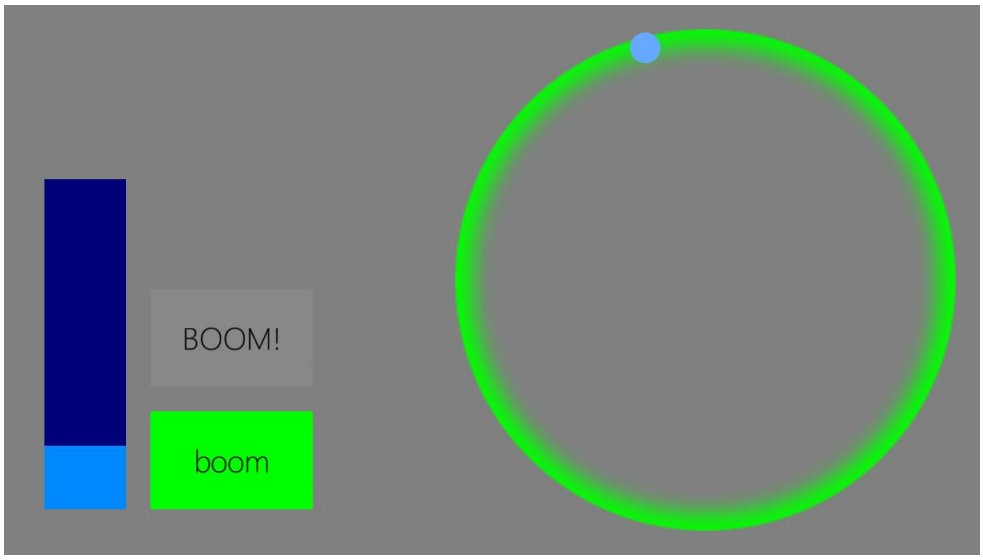

Figure 5.2: The rocket controller smartphone screen.

As mentioned above, the players' task is to build defensive structures commonly referred to as towers in order to prevent the enemies from reaching the end of their path. In order to build these towers a certain amount of game currency, referred to as gold, is needed. All towers are steam powered and need to have a certain pressure built up in order to function. When a tower's pressure has run too low, it needs time to rebuild its pressure before being able to function again. The towers are divided into different types, each with a unique way for the user to interact. When a user assumes control of a tower, the client transitions into a controller mode giving access to special operations that can be performed on the tower. One such transition is presented in figure 5.2 which shows the controller state for the missile tower. The player can use the big circle to aim the rocket at different parts of the game map, and uses either the "boom"-button to fire a small missile or the "BOOM!"-button to fire a large missile.

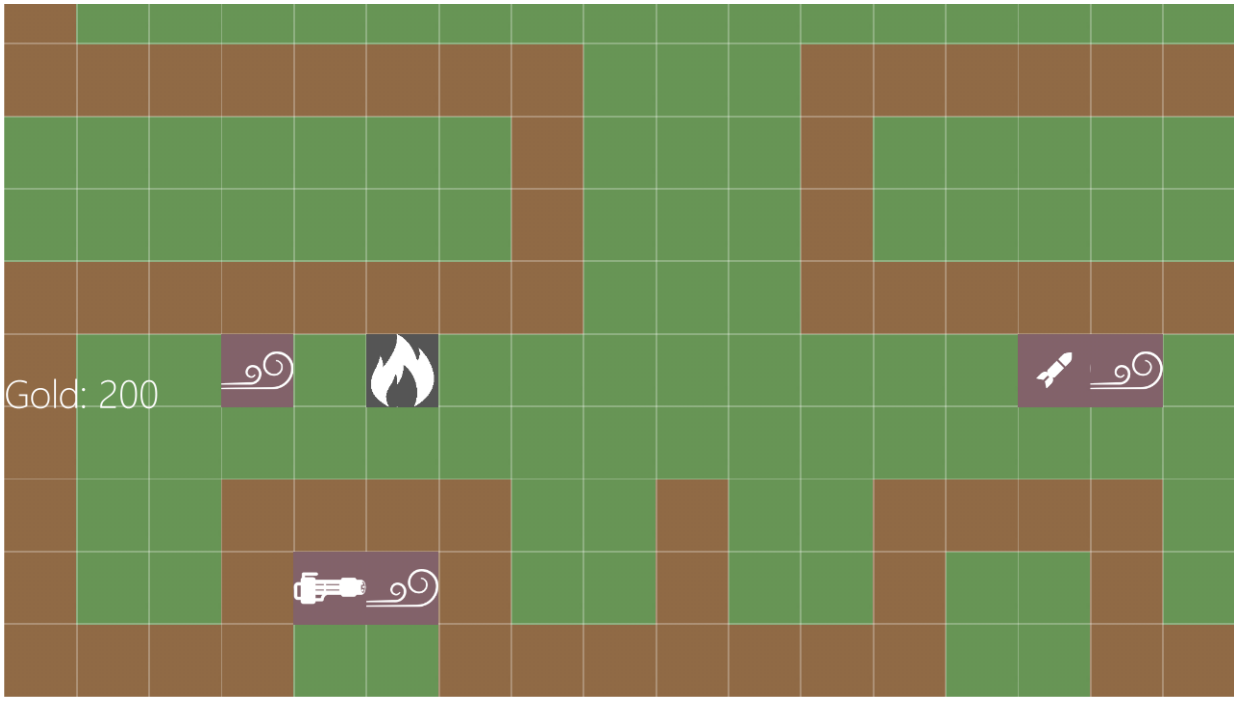

Figure 5.3: The client's building screen. Green squares are buildable, while brown ones are not. The different symbols denote different types of towers.

In figure 5.3 we can see the building screen on the phone. While in this mode the user can build and enter towers. The user builds a tower by tapping the tile on the gridmap where the tower should be built. A selector is then displayed where the user can select the type of tower to be built. By clicking on a tower type in the selector another selector is displayed where the user is given the possibility to upgrade the tower to make it more powerful or to assume control of the tower.

The game includes several types of enemies, each having different values for parameters such as health (how hard it is for the towers to shoot them down), speed (the speed at which they traverse the path), as well as a set of special abilities such as a temporary increase in speed or the ability to replenish health. Figure 5.4 shows the server portion of the game, with multiple enemy types on the path in the lower left of the picture.

Figure 5.4: The game as seen on the server.

5.3 Pictionary

Pictionary is a classic board game where one player has access to information that is unavailable to the other players. This is called information hiding which was discussed in section (1.1). Pictionary was also implemented to test the systems viability in games that are not played in real time.

One player is chosen at random to draw a picture representing a randomly chosen word. The drawing player uses her smartphone as a canvas, and the main screen mirrors what she draws. The remaining players guess what word is being drawn. If a player guesses correctly, both the guesser and the drawing player get one point. After a certain amount of time a new player is assigned to draw and a new round begins. Whoever has the most points after every player has been assigned to draw a particular number of times, wins the game. Figure 5.5 shows a session of Pictionary. As can be clearly seen the drawing player is painting the word “football”.

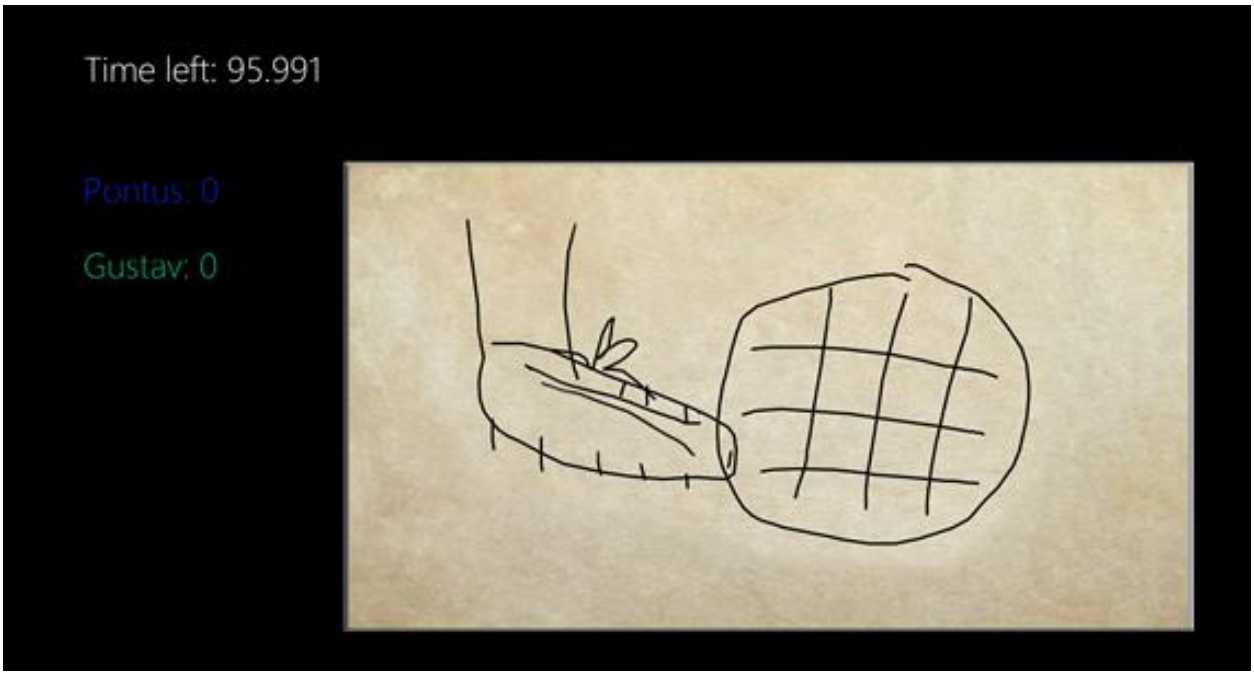

Figure 5.5: Pictionary as seen on the server. Two players are engaged in a session.

5.4 Questionary

Questionary is a classic trivia game, in the spirit of Trivial Pursuit. This game was built to provide a more complete set of games. It does not test any special functionality not present in the other three games.

Concept

Questionary is a trivia game where the goal is to correctly answer questions in six categories. A correct answer is rewarded with a medal in the relevant category. When a player has collected three medals in each category, she wins the game. The questions to answer are globally visible to all players which means that more than three questions in a category might be correctly answered by the same player. To give the players incentive to still try and answer correctly after getting three correct answers in a category, the player gets bonus points. These bonus points can be used to purchase medals in categories in which the player has not already received three medals.

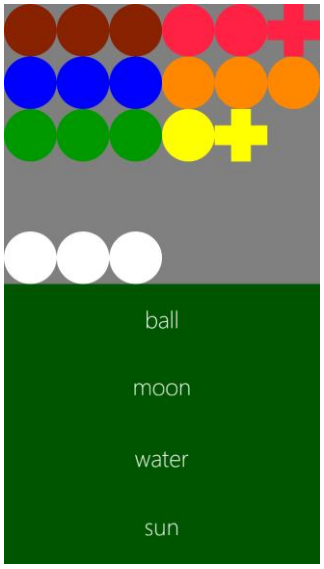

Figure 5.5: The client of Questionary. The lower half of the screen is used to select different possible answers to the current question, and the upper half is used to display the points the player has collected in every category. The white bonus points are exchanged for points in other categories by tapping one of the plus signs.

The question to be answered is displayed on the public computer screen and each player is presented with four alternatives to choose from on their smartphone. Every player selects the alternative they think is the correct answer. As soon as every player has made a choice, or the time limit has been reached, points are awarded to each player who selected the correct alternative. Afterwards, the correct answer is shown on the public screen.

6. Future work

This chapter concerns possible future work, which if pursued will make the platform even more appealing to consumers and developers.

6.1 More platforms

To make the system available to a wider audience, more platforms need to be supported. Most urgently, the application should be ported to the iOS operating system on the mobile side, and Mac OSX operating system on the server side.

6.2 User testing

In order to evaluate the functionality of the system as a whole and to evaluate the methods of controlling the created comparative player testing needs to be carried out. This type of testing was unfortunately not carried out due to time constraints. The first game described in this paper, *Achtung*, is very well equipped to test and evaluate different control schemes such as using the accelerometer or using two buttons for turning left or right, as seen in the research of Joselli et al [10].

6.3 RCE sandboxing
The current system is not taking any security measures. The implications of this are that a malicious developer may execute arbitrary code on the connected smartphones, which of course cannot be tolerated. To consolidate this flaw, RCE sandboxing can be used.

6.4 Resource version control
To make the system more effective, a version control mechanism can be implemented. This would allow the server to query the file system of each connected smartphone and only send the game assets that are not already present.

6.5 Lower latency network
The latency testing done shows that both UDP and TCP work fairly well for soft real time applications. However, in rare circumstances (less than (1%)) there is an unacceptable amount of latency between two consecutive sensor updates. More research needs to be done to investigate if it is possible to remove even this latency using a different network configuration or different network technologies.

6.6 Marketability
It can be said that any game has three sets of features that indicate how close it is to being done and marketable. These three sets are core features, required features and desired features.
The core consists of features and mechanics that define the game and distinguish it from other titles, just the core features will not constitute a marketable product. The required features are what creates consistency in a game and makes a game playable. However, in order to create a commercially competitive product the desired set of features need to be included, these are what gives a game a polished and complete feel [52]. The games accompanying our system currently cover the set of core feature and a majority of the required features. However, features from the desired set have been omitted due to time constraints. In order to make the system marketable, features in all sets are needed.

7. Discussion
The number of people who own smartphones is growing at a fast pace, and has been projected to reach 5.6 billion in 2019 [1]. Smartphones include many advanced hardware capabilities that can be utilized in a large number of ways. This project has aimed to make use of these devices to the fullest extent possible, by creating a virtual game console which uses smartphones as controllers.

A virtual game console that uses smartphones as controllers have many advantages over conventional gaming consoles such as the Xbox and the Wii U. Since the players can use their smart phones to control games, there is no need to purchase any additional controllers. This is usually required if people are to play together on a traditional console, since such consoles usually only come with one controller. Furthermore, smartphones contain hardware which is overly expensive to use in conventional video game controllers. Buying four smartphones only to use them as video game controllers would be absurd, due to their steep purchase prices. However, the reality of the situation is that not only do most people already own a smartphone, they carry it with them at all times, ensuring that all players are able to join in. In our system there is no set limit on the number of players that can join in. This makes it an ideal platform for social activities where many players are in the same room.

There are also drawbacks with this approach. This is mainly due to the fact that smartphones lack many important hardware components present in modern day controllers. These include things like thumb sticks and hardware buttons. More avid players have come to expect these on a game controller making the transition awkward. However, there are many players who only play games on their smartphone and these players will have no problem using them on this platform.

All players interact with the console through their smartphones. This enables a whole class of games which previously was not playable digitally on a local level. Specifically, the players can hide information from each other. An example of this would be Texas Hold'em, where the main attraction is the suspense of not knowing what cards your opponents have been dealt. Due to prioritizing different games we unfortunately did not have time to make a game which is played in this way.

During the course of the project, we have all become polyglot programmers. This is more or less required if we are to make a game spanning multiple platforms which only allow certain languages to be used. However, most of these languages are very similar to C which makes it simple to swap between them. The mental model does not have to change very drastically. The exception to this is the Lua programming language which due to its dynamic nature differs significantly from the other languages.

Working with the D programming language allowed us to create complex code generating systems with little effort, which boosted our development process considerably. Defining most variables in SDL documents has led us to program in a very data driven fashion, being able to quickly tweak individual fields and properties without having to recompile the application. Similarly, content reloading has made it possible to change textures and fonts and instantly see the result in the game.

Using a single smartphone client to run multiple different games has been a very rewarding process. Getting a game up and running is just a matter of copying a few files from another project. There is no longer any need to recompile another application on the phone and all the headaches that comes along with it.

The protocol and design decisions made for the network part of the system has in retrospect worked out well. The network has evolved alongside the applications during the entire project. We built the entire network from TCP and UDP sockets, and have arrived at a functional system with an acceptable level of latency for all of our games, which is quite the achievement. Nevertheless, there are still things that can be done to improve the network. Further usage of the network will reveal its weaker points.

The platform we have described is very comprehensive. It has most of the functionality one would expect of such a system. However, like any software it has room for improvements. The code would benefit from further revision. It is not close to a state where it can be released to the public and used by others as a library.

There have been several obstacles to overcome during the project. Most of these originated from the Android operating system. The activity lifecycle in particular was a major obstacle. The Android platform is also very fragmented, meaning that different things work in different ways on different devices. For instance, one of our test phones, the HTC One, does not support rendering of non-POT (power of two) textures. This peculiarity was not well documented, and hence it took us quite some time to narrow it down to that particular deficiency.

The games we have created cover only a small part of the entire interaction space present between phones and computers. We have tested information hiding, accelerometer usage, and a large number of touch screen controls. This is just scratching the surface of what is possible. More testing has to be done in order to find what works and what does not.

We are very pleased with the performance of the applications. We lowered the power usage of the phone application by an order of magnitude, without dropping too many frames. Throughout the project performance has played a big part in how the code is structured and we think we did a reasonable job at this.

There are always problems associated with creating something new and the first version of something is seldom perfect. If we apply lean development methodologies, what is presented in this report can be seen as the minimum viable product for a system of this kind. As has been shown in the previous chapters, the system is already usable and has potential to grow. However, we are quite pleased with how the system turned out, given the difficulty of the problems that needed to be solved and the limited timeframe given to solve these problems.

8. References

[1] Ericsson. (2013, November) Ericsson Mobility Report. [Online]. http://www.ericsson.com/res/docs/2013/ericsson-mobility-report-november-2013.pdf

[2] Caroline J. Kistin, Barry Zuckerman, Katie Nitzberg, Jamie Gross, Margot Kaplan-Sanoff, Marilyn Augustyn and Michael Silverstein Jenny S. Radesky, "Patterns of Mobile Device Use by Caregivers and Children During Meals in Fast Food Restaurants," *Pediatrics*, Mar. 2014.

[3] Sean Buckley. (2011, June) engadget. [Online]. http://www.engadget.com/2011/06/18/nintendo-says-one-wii-u-controller-per-console-robs-player-two/

[4] Microsoft. (2014, May) xbox.com. [Online]. http://www.xbox.com/en-US/smartglass

[5] Timothy B. Smith, J. Bradley Layton Julianne Holt-Lunstad, "Social Relationships and Mortality Risks: A Meta-analytic Review," *PLoS Medicine*, vol. 7, no. 7, July 2010.

[6] Google. (2014, May) android developers. [Online]. https://developer.android.com/about/dashboards/index.html?utm_source=ausdroid.net

[7] Google. (2014, May) android developers. [Online]. http://android-developers.blogspot.se/2010/12/android-23-platform-and-updated-sdk.html

[8] F. Ferreira dos Santos, S. Rodrigues dos Santos S.M. Malfatti, "Using Mobile Phones to Control Desktop Multiplayer Games," in *Brazilian Symposium on Games and Digital Entertainment (SBGAMES)*, Florianopolis, 2010, pp. 230-238.

[9] Google. (2014, May) play.google.com. [Online]. https://play.google.com/store/apps/details?id=com.pocketappbuilders.androidpcgamepad

[10] J. R. da Silva, M. Zamith, E. Clua, M. Pelegrino, E. Mendonca, E Soluri M. Joselli, "An Architecture for Game Interaction using Mobile," in *2012 IEEE International Games Innovation Conference (IGIC)*, Rochester, NY, 2012, pp. 1-5.

[11] B.C. McDonald, S.J Warden, J. Yonkman, A.J Saykin, B. Shirley, M. Huber, B. Rabin, M. AbdelBaky, M.E. Nwosu, M. Barkat-Masih, G.C. Burdea M.R. Golomb, "In-hme virtual reality videogame telerehabilitation in adolescents with hemiplegic cerebralpalsy," in *Archives of Physical Medicine and Rehabilitation vol. 91*, 2010, pp. 1-8.

[12] M. McNeill, D. Charles, P. Morrow, J. Crosbie, S. McDonough J.W Burke, "Serious games for upper limb rehabilitation following stroke," in *Conference in Games and Virtual Worlds for Serious Applications*, Coventry, 2009, pp. 23-24.

[13] S. Nestler, A. Dippon, G. Klinker F. Echtler, "Supporting casual interactons between board games on public tabletop displays and mobile devices.," in *Personal and ubiquitous computing*, 2009, pp. 609-617.

[14] J. Koskela, K. Ollila, S. Mäki, R. Kulpa-Bogossia, T. Heikkinen, T. Ojala P. Luojus, "Wordster: collaborative versus competitive gaming using interactive pubic displays and mobile phones," in *Proceedings of the 2nd ACM International Symposiumon Pervasive Displays*, 2013, pp. 109-114.

[15] P. Coulton, W. Bamford, R. Edvards T. Vajk, "Using a mobile phone as a "Wii-like" Controller for Playing Games on a Large Public Display.," in *International Jurnal of*

Computer Games Technology, 2008, pp. 1-6.
[16] Hannes Holm, Mathias Ekstedt Theodor Sommestad, "Estimates of success rates of remote arbitrary code execution attacks," *Information Management & Computer Security*, vol. 20, no. 2, pp. 107-122, 2012.
[17] Anthony Grimes et al. (2014, May) tryclojure. [Online]. http://tryclj.com/
[18] Luiz Henrique de Figueiredo, Waldemar Celes Roberto Ierusalimschy. (2014, May) Lua 5.1 reference manual. [Online]. lua.org/manual/5.1/manual.html
[19] Mike Pall. (2014, May) LuaJIT performance: arm. [Online]. www.luajit.org/performance_arm.html
[20] Mike Pall. (2014, May) LuaJIT compatability. [Online]. luajit.org/luajit.html
[21] Philippe Sigaud. (2012, January) Phillippe Sigaud github d-templates. [Online]. https://github.com/PhilippeSigaud/D-templates-tutorial/blob/master/D-templates-tutorial.pdf
[22] Andrei Alexandrescu, *The D Programing Language* , 1st ed. Boston, United States of America : Addison - Wesley Professional, 2010.
[23] Mike Parker. (2014, May) github derelict aldacron. [Online]. https://github.com/aldacron/Derelict3
[24] Camilla Berglund. (2014, May) glfw. [Online]. http://www.glfw.org/
[25] Hervé Drolon. (2014, May) FreeImage. [Online]. http://freeimage.sourceforge.net/
[26] Stephane Peter, and Ryan Gordon Sam Lantinga. (2014, May) SDL_mixer 2.0. [Online]. http://www.libsdl.org/projects/SDL_mixer/
[27] The Khronos Group. (2010, Mars) Opengl 3.3 specification. [Online]. http://www.opengl.org/registry/doc/glspec33.core.20100311.pdf
[28] Google. (2014, May) Android NDK. [Online]. https://developer.android.com/tools/sdk/ndk/index.html
[29] The Khronos Group. (2010, November) OpenGL es 2.0 Full Specification. [Online]. http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf
[30] Oracle. (2014, May) JNI Specification. [Online]. http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html
[31] Google. (2014, May) Android SDK. [Online]. http://developer.android.com/sdk/index.html?hl=sk
[32] NVidia. (2014, May) NVidia android lifecycle. [Online]. http://developer.download.nvidia.com/assets/mobile/files/AndroidLifecycleAppNote_v100.pdf
[33] Digitalmars. (2014, May) DMD garbage collector. [Online]. http://dlang.org/garbage.html
[34] Cem Cebenoyan, "Chapter 28. Graphics Pipeline Performance," in *GPU Gems*, Randima Fernando, Ed. United States of America: Addison-Wesley Professional, 2004, ch. 28.
[35] NVidia. (2014, May) NVidia Improving batching Using Texture Atlases. [Online]. http://developer.download.nvidia.com/SDK/9.5/Samples/DEMOS/Direct3D9/src/BatchingViaTextureAtlases/AtlasCreationTool/Docs/Batching_Via_Texture_Atlases.pdf
[36] Roberto Ierusalimschy. (2014, May) Lua performance tips. [Online]. http://www.lua.org/gems/sample.pdf
[37] Yu-Wei Su, Chung-Chou Shen Jin-Shyan Lee, "A Comparative Study of Wireless Protocols: Bluetooth, UWB, ZigBee, and Wi-Fi," in *Industrial Electronics Society, 2007.*

IECON 2007. 33rd Annual Conference of the IEEE, Taipei, 2007, pp. 46 - 51.

[38] W. Richard Stevens Kevin R. Fall, *TCP/IP Illustrated Volume 1: The Protocols*, 2nd ed., Brian Kernighan, Ed. United States of America: Addison - Wesley Professional, 2011.

[39] Jeffrey Mogul. (1984, October) RFC922. [Online]. http://tools.ietf.org/html/rfc922

[40] Y. Rekhter et al. (1996, February) RFC 1918. [Online]. http://tools.ietf.org/html/rfc1918

[41] John Nagle. (1984, January) RFC 896. [Online]. http://tools.ietf.org/html/rfc896

[42] Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, Mark Claypool Tom Beigbeder, "The Effects of Loss and Latency on User Performance in Unreal Tournament 2003," in *SIGCOMM'04 Workshops*, Worcester, 2004, pp. 144 - 151.

[43] Ecma international. (2013, October) Ecma international - Json standard. [Online]. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[44] W3C. (2008, November) XML - specification. [Online]. http://www.w3.org/TR/2008/REC-xml-20081126/

[45] Bitsquid. (2014, May) Bitsquid resouce page. [Online]. http://www.bitsquid.se/files/resource_management.html#Simplified

[46] Digital Mars. (2014, May) Dlang attribute documentation. [Online]. http://dlang.org/attribute.html

[47] Microsoft. (2014, May) MSDN - XNA. [Online]. http://msdn.microsoft.com/en-us/library/bb200104.aspx

[48] Lode Vandevenne. (2014, May) LodePng. [Online]. http://lodev.org/lodepng/

[49] Andreas Jönsson. (2014, May) Angelcode. [Online]. http://www.angelcode.com/products/bmfont/

[50] BrashMonkey. (2014, May) Spriter. [Online]. http://www.brashmonkey.com/spriter.htm

[51] Unity Technologies. (2014, May) Particle system documentation. [Online]. https://docs.unity3d.com/Documentation/Components/class-ParticleSystem.html

[52] Julian Gold, "Open-ended design," in *Object-oriented Game Development*. Harlow, England: Addison-Wesley, 2004, ch. 2, pp. 18-19.