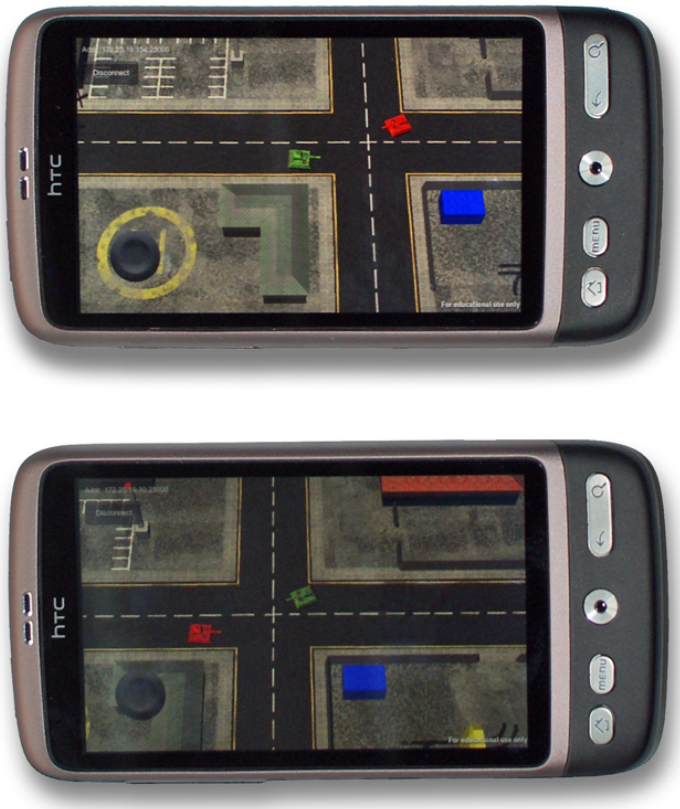
TileShifter



A Peer-2-Peer Multiplayer Game for Smartphones

Master Thesis during DAT6 by
— Group D601a —
Jais Heslegrave & Thomas Justesen

July 29, 2011 at Aalborg University

AALBORG UNIVERSITET

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
Phone +45 9940 9940
Fax +45 9940 9798
http://www.cs.aau.dk

Student Report — Master Thesis

Titel:
TileShifter
— A Peer-2-Peer Multiplayer Game for Smartphones...

Theme:
Peer-2-Peer Multiplayer on Mobile Devices, Distributed Systems

Project period:
DAT6, spring 2011

Project group:
D601a

Authors:
Jais Heslegrave
Thomas Justesen

Supervisor:
René Rydhof Hansen

Printcount:
4

Nr. of pages:
77

Appendix:
A cd with source, executable, testresults and simulator results.

Completed & signed:
July 29, 2011 at Aalborg University

Abstract:

Most people have tried to be bored, while waiting for something, whether it is a bus or a person late for a meeting.

Since Smartphones began taking over the mobile phone market, more and more people tend to use their Smartphones to fill periods of boredom, with either business — such as reading and responding to mail, or with entertainment — such as games. Due to the rapidly expanding market of Smartphones and the increase in number of interested consumers, more and more apps for Smartphones are developed. It does, however, seem that in the area of games, very few apps support multiplayer, especially for larger groups of people.

TileShifter is the product of this project, and is an attempt at developing an action-packed, scalable, Peer-2-Peer multiplayer game. The aim is to prove that, although the resources on mobile phones are low, Smartphones are at a stage, where scalable multiplayer is possible. The solution for TileShifter is to introduce a custom set of algorithms, which correlates well with the game design and a virtual world of tiles.

Contents

Preface

This report has been written during the DAT6-project period and is a Master Thesis by group D601a at Aalborg University. The theme is "Peer-2-Peer Multiplayer on Mobile Devices, Distributed Systems".

References to sources are marked by [ABc#], where ABc# refers to the related literature in the bibliography at the end of the report.

The appendix to the report is found as test results, simulation results, source code and executables on a compact disc, located on the very last page of the report.

The entire report is written in English and no translation will be accessible. Abbreviations and acronyms are at first appearance written in parentheses, to avoid breaking the reading stream. The report is written in LaTeX and is accessible as a PDF-document.

A special thanks to Rex Nebular (Morten Lund Søegaard) for supplying the game with sound and music, our peers for constructive peer reviews of this text, Unity Technologies for providing educational licenses and Aalborg University for providing Smartphones.

Signatures:

Jais Heslegrave

Thomas Justesen

Chapter 1

Introduction

Games on Smartphones are already soaring in popularity, but why play with yourself when you could be playing with your friends. In this report, the game TileShifter is introduced, which lets Smartphone owners play against each other, in an exciting tank combat game. TileShifter uses Peer-2-Peer networking, to allow many players to play simultaneously, and drop in and out of the game on a whim.

In the past few years, the Smartphone sales have risen noticeably according to Gartner [PS11]. This trend has lead to the development and production of increasingly powerful devices, both in terms of processing power and in the number of features. The introduction of Apple's App Store opened a market for numerous new software and game developers for Smartphones, both independent and professional. According to recent news from Apple [PM11], a reason for this success could be the ease of development and publishing of the apps. It means that any developer with an idea and time, is capable of creating and distributing an app to the rapidly expanding app store market. The App Store concept was quickly adopted by other Smartphone developers, who sought to reap the benefits.

As Smartphones increase in computational power, more and more elaborate and demanding games appear in the app stores. It is clear that Smartphone games can be successful, demonstrated for instance by "Angry Birds" recently reaching 100 million downloads across multiple platforms [Lei11]. It seems, however, that the game developers for Smartphones are thinking mostly in single player games. Few multiplayer games exist in the stores, and those that do exist are either not real-time collaboration (such as "Pass and Play" and turn based games) or limited to very few players (i.e. 2 — 4 players). One can only speculate why so few multiplayer games are available, since the Smartphone is more than capable of network multiplayer games, with the built-in network

interfaces.

This project tries to discover, that if the right game design choices are made, it is possible to create a scalable multiplayer game for Smartphones, with wireless connectivity and sufficient computational power. To discover this, TileShifter is developed. TileShifter is a scalable multiplayer action game, that uses a Peer-2-Peer architecture in connection with custom-tailored tile algorithms.

In the Analysis Chapter, the initial part discusses a few interesting game concepts, that could be implemented. Afterwards the Chapter defines the term Smartphone, leading to which features are available in a Smartphone. The focus of the report is on networking, so an analysis of possible physical network connections, protocols and architectures available on a Smartphone, leads the networking design of TileShifter in the right direction.
In the Design Chapter, the game design of TileShifter is explained, along with how game design choices are used to influence the network connectivity, using a number of custom tile-based algorithms. These algorithms are detailed in depth, along with a description of the wireless network test and algorithm simulation design.
Special network related implementations in TileShifter are exemplified in the Implementation Chapter, along with interesting examples from the tile algorithm implementations.
Evaluations of TileShifter, the tile algorithms and the wireless test results are brought forward and elaborated upon in the Evaluation Chapter. Finally, the Evaluation Chapter describes further possible developments on all aspects of TileShifter, and concludes on the project.

1.1 Problem Statement

The goal of this project is to examine the possibilities of developing a Peer-2-Peer game for a Smartphone environment. This requires designing a game fitted for the Peer-2-Peer model, as well as investigating the capabilities of modern Smartphones. The game is intended to be quick fun for a group of players joining together to play, as well as a game that is quick to get into the action. Therefore this project will focus on local multiplayer using the wireless connection. Letting any player leave or join at a whim, still maintaining a stable game experience for the remaining players and letting the game scale up to a certain amount of players, is of importance. A relevant goal is furthermore, to examine how game design, networking and program design can mesh together to enhance and shape the multiplayer experience.
These thoughts coalesce into one problem, that is tackled during this project: *"Can a multiplayer Peer-2-Peer game be developed for a Smartphone platform?"*.

The game should try to live up to as many of the Peer-2-Peer system characteristics as possible, namely: Each peer contribute resources, all peers should be equal, no server or tracker is necessary for the system, anonymity for peers and an efficient algorithm should handle data and workload distribution [CDK05]. These problems can be formalised into the following requirements:

Peer-2-Peer The game must be Peer-2-Peer and should not require a dedicated server or tracker of any kind.

Smartphone Platform The game is to be developed primarily for the modern Smartphone platform, capable of executing the chosen game engine.

Scalability A certain amount of players should be able to join the game and have an enjoyable experience, and every player joining should contribute in some way.

Robustness Players should be able to join and leave on a whim, with minimal impact on the remaining players.

Drop in/out Gameplay Game pacing should allow players to have a fun experience immediately upon joining.

Game Driven Design As many networking features as possible should contribute to the gameplay in a meaningful way.

In addition to these requirements, some delimitations also exist, namely in the area of security and NAT. As such the following delimitations have been formalized:

Security Anti-cheating, anonymity and security are whole research areas in themselves, and thus will not be the focus of this project. In addition the game is meant to be played locally, by players in the general vicinity of each other, this means the players themselves are the anti-cheat mechanism.

NAT Network Address Translation is an obstacle for every networked application attempting to use the Internet. The game is meant to be played locally in this phase of development, not over the Internet. As such, NAT and Internet communication is no concern.

Chapter 2

Analysis

Before designing the game and network connection, the Analysis Chapter aims to examine relevant existing solutions to related problems. The purpose is to generate a knowledge base, satisfactory enough to create the multiplayer game, TileShifter.

Among the important subjects discussed in the analysis phase are, existing games on the market for Smartphones and what game concepts have been contemplated in this project. The analysis sheds light over what possibilities the modern Smartphones hold. A very important part of this project is the networking, and this Chapter also details the different paths to follow and which is chosen. Finally the Chapter explains the choice of using Unity3D for the development.

2.1 Game Concepts

The way to acquire games for a mobile device is using using the various app stores that support them, whether an Apple or Android device. Examining both the App Store [App11] and Android Market [Goo11] reveals some of the multiplayer games available, that are popular with users. Many of the multiplayer games are of the turn-based variety, where players take turns making moves in the game, and as such do not play at the same time. This allows the game to be played in a "Pass and Play" manner, also known as "Hotseat", where players take turns using the device. This approach is not suitable for an action based game. Turn-based games also commonly support multiplayer with multiple devices, this can be locally over Bluetooth or wireless network, but also online using wireless network or 3G. Games like these include Checkers, Chess, Risk and Monopoly. Multiplayer real-time games for the mobile

devices can be found supporting both local and online play, most commonly these games support 2-4 players. Rarely do they support upwards of 16 players at once. Some games are of the MMORPG variety, which offer online play for many players at once, these games do require servers however, and cannot be played locally.

In order to investigate the possibilities of scalable Peer-2-Peer games on a mobile platform, a suitable game concept has to be chosen to develop a reasonable, yet simple, game design. The game design must be basic enough to not obscure key issues, yet advanced enough to be an interesting game. These concerns lead to several different game genres that could be pursued for the developed game, these include: action games, puzzle games, turn based games, arcade games, trivia games and more. These are also game types that lend themselves to the mobile platform, given that some game types are difficult to properly control on a touch screen interface. The following game concepts have emerged from these considerations:

Tanks The tanks idea is inspired by the classic games of tanks shooting at each other. This basic idea is easily expandable, and the simple mechanics of tanks driving around, shooting and destroying each other would be preserved. The unique twist here is that the arena that tanks would be fighting in would be made of square building blocks that could fit together and easily expand the playing field to allow for more and more players to combat each other. A concept sketch of the Tanks game is depicted in Figure 2.1.

Board Game This idea hits a mark near turn based game and arcade game, being a game based on the board game "Snakes and Ladders". This game idea would have players take turns to roll a die to advance up the game board and try to hit ladders while avoiding snakes. Players could pick up power-ups along the way, to influence the gameplay with extra moves or obstacles.

Quiz A basic quiz trivia game where the players have to answer questions or solve puzzles competing against each other. The game could feature a variety of questions, ranging from simple multiple choice questions to writing answers on the phone to recognizing shapes. Points would be awarded for correctness and speed.

Arcade Game An idea based on a tabletop game which is a hybrid between an action game and an arcade game. Players take turns revealing cards on their hand bearing shapes and colours, if a similar shape was already in play in an opponents revealed cards, these two players would race for a totem, and the winner of this duel would give his revealed cards to the loser. The player who discards all cards first, wins.

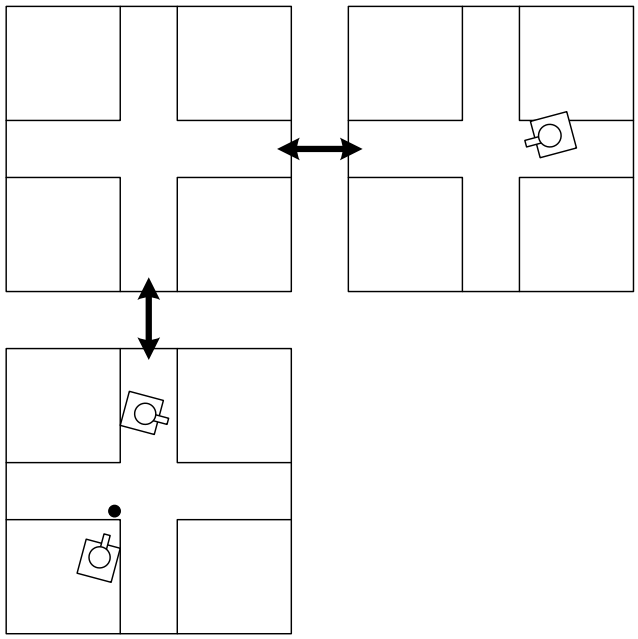


Figure 2.1: A concept sketch of the Tanks game.

After some deliberation on which game concept to run with, the final decision ended up being the "Tanks" concept. It is named "TileShifter", based on the concept of the square tiles that the playing field would be built of. This is due to the tile idea allowing an interesting approach to the Peer-2-Peer aspect of the multiplayer game, a more detail explanation of this idea is found in Section 2.3.2 about network protocols and architecture. Each peer controlling one of these tiles and connecting to other peer tiles to play, this game concept can also be made serverless, which is one of the goals of this project. The tile— and game design concepts are further detailed and examined in Section 3.1. The other game concepts do not have these opportunities to integrate interesting game concepts with networking and game design as the tank concept did. In addition the other games were more turn-based and it seemed as though turn-based games were the norm, as examined earlier. Pursuing an action game approach was more appealing, since it meant heading in a more unique direction.

2.2 About Smartphones

There is much talk about Smartphones at the present day, because more and more consumers have an increasing interest in their features, and the Smart-

phone developers respond to the rising interest by adding more features and increase the complexity and processing power. It is a good idea to define what is meant by the word Smartphone before initiating the development (i.e. when can a mobile phone be called a Smartphone). No official definition of the word Smartphone exists, probably because it is used in the industry as a buzzword to increase sales of mobile phones. Some even try to define a Featurephone as a phone with lesser features than a Smartphone, to add more depth to the discussion [NZ09]. Small attempts at definitions can be seen in *PCMAG.com* [PCM11] and in 2010 Steve Litchfield tries to define the term on *All About Symbian.com* [LS10] as well. In the context of this project, the word Smartphone is used, not as a sales argument, but as a specific type of consumer device, so the definition must specify a set of necessary features needed to play a multiplayer game on the device.

2.2.1 Defining A Smartphone

In the context of this project, the Smartphone must be able to run a 3D graphical game and have a wireless network connection. Therefore the definition of a Smartphone in connection with this report, is a phone capable of the following:

- Resources to be able to run 3D applications.
- Installation of applications and games.
- Network communication through wireless connections.

This is a rather loose definition of a Smartphone, but it is viewed as sufficient for the scope of this project.

2.2.2 Features of The Smartphone

As mentioned, Smartphones have more and more features and the computational power increases with each new product that arrives on the market. The Smartphone is becoming a possible substitute for the computer or laptop. Often more functions are available in Smartphones than in most laptops and PCs, such as assisted GPS, multitouch displays and accelerometer. A typical Smartphone runs with an underclocked processor of 1 GHz, with 256 — 512 MB RAM and very varied storage capacity. The reason for the processor underclocking is mainly to save power and reduce heat of the device [CE10].

The specifications and capabilities of the Smartphones, and the fact that users have them at hand at any time of day, makes them ideal as small gaming platforms. Either for small quick games while waiting for a bus, or for larger

games for longer idle periods. The games could easily be used in a social connection, when using the built-in wireless networking or Bluetooth connections. The games can be made for the different platforms to be able to communicate across platforms. This will reach a larger group of consumers. Another idea is to utilize some of the built-in features such as GPS and the accelerometer, to enhance and enrich the game experience.

2.2.3 Choice of Smartphones

At the moment three major Smartphone developer platforms are available, the Blackberry by RIM, the iPhone iOS by Apple and the Android by Google. Blackberrys are made solely for business and is not a good gaming platform due to the smaller screen and market share. iOS and Android on the other hand, are very good choices for the game. Both platforms are relatively simple, but to develop for the iOS platform, Apple requires a one year developer license as well as a Mac computer to compile the application. Google Android is much more lenient in this regard, as no special licenses or equipment are required to build an application to a device. These facts mean that Google Android is the chosen platform for this project.

The final choice for the development and test platform is two HTC Desire Smartphones, with 1 GHz floating point processor, 512 MB RAM, wireless networking 802.11b/g and Android operating system.

2.3 Smartphone Networking

When developing a network multiplayer game, it is imperative to analyse the possibilities for network connections and network architecture. These are two different subjects within networking for multiplayer games. The network connections refer to the hardware components used for the network connections, and the network architecture refers to protocols, while relations between the participating client devices.

In this Section, the aim is to describe the networking hardware, architecture and protocols, which can aid in the development of an application for the HTC Desire. The Section concludes with the choice of networking components and the motivation for this choice. The first step, in Section 2.3.1, is to decide upon what physical network connection is paramount to the project, while in Section 2.3.2, different networking protocols and architectures are discussed for use in the project.

2.3.1 Physical Network Connection

In Section 2.2.3 the HTC Desire was chosen as the target device for this project. This Section therefore uses the Desire as a reference for what hardware is available on the Smartphone for networking communications.

In the following Sections, the positive and negative sides in relation to multiplayer network development are discussed for: Bluetooth, the cellular data and the wireless network, to determine the best choice for this project.

Bluetooth

Bluetooth is a widely used wireless proprietary — but open — technology. Its purpose is to connect and exchange data between devices over short distances. Bluetooth was developed by Ericsson [Blu11a, KRA08] and it is maintained by the Bluetooth Special Interest Group, mainly consisting of members from the telecommunication industry [Blu11b].

Bluetooth in mobile devices is often used to transmit sound to a headset, or connect the device to a computer to work as an internet modem. Because of this, many people may forget or be unaware, that Bluetooth is capable of more than a single connection between two devices. Eight devices can be connected, where seven of them act as slave devices, and a single device is the master, this is called a Piconet [KRA08, MS99]. One of these devices in this Piconet could be connected to another Piconet forming a Scatternet, which means that it is possible to create a larger network using Bluetooth. See Figure 2.2 for an example of two Piconets forming a Scatternet. The Bluetooth antennas are omni-directional, meaning that the signals go in all directions, and do not need to be directed, signals can also go through obstacles. Finally, the Bluetooth applications can be both synchronous and asynchronous. In the dissertation "Design and Deployment of Wireless Networked Embedded Systems" by Jan Beutel [Beu05], experiments with large scale Scatternet using Bluetooth is possible for a network of 70+ connected devices.

Speaking against Bluetooth as a solution for TileShifter, is that the connection speed is limited to theoretically 3 Mbit/s, which is 375 kB per second, at best in Bluetooth v.2.1, which is used in the HTC Desire[1] [HTC11]. This speed is expected to be a bit low for a near real-time multiplayer network game. Furthermore, the Piconets and Scatternet will be difficult to maintain when players leave the network, and will result in much work to create a solution that keep Piconets running if masters leave. The distance limitations

[1] Bluetooth v.2.1 with Enhanced Data Rate. Never versions of Bluetooth v.3.0 and 4.0 is said to have a Data Rate of 24 Mbit/s

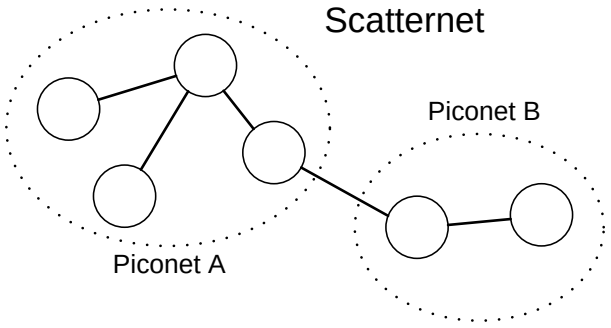


Figure 2.2: A concept sketch of a two Bluetooth Piconets forming a Scatternet.

of approximately 5 — 100 meters, depending on if it is a Class 1, 2 or 3 Bluetooth connection, is viewed as a minor factor.

GSM, GPRS, EDGE & 3G

Smartphones can directly access the internet at all times, this is in modern days done using a built-in 3G network connection which is the third generation of mobile internet communication, also known as High-Speed Packet Access (HSPA). 3G mobile data networking is a result of the demand of higher data rates on the mobile devices [IML+03] and is said to be able to perform at a theoretical speed of 42 Mbit/s (approximately 5 MB per second), while Danish companies promise to deliver up to 32 Mbit/s [Dan11] and up to 100 Mbit/s using the slowly expanding long term evolution 4G [Tel11]. The speed depends on several factors, such as obstacles between the device and antenna, the number of concurrent users and the distance between the device and the antenna. A more realistic data rate is about $1 - 10$ Mbit/s.

On the downside there is no way of controlling the routing on a mobile data network such as 3G, EDGE, GPRS or GSM internet connection. The device does get an IP address on the network, but ports are closed and connections from device to device are highly limited. This is mainly to secure the devices from hacking and other types of misuse of the network. The fact that there is no guarantee that the devices will be able to communicate using this network connection, leaves it to be an unreasonable choice for this project as well. Communication could be established using advanced NAT punchthrough mechanisms, in which the concept is to "punch" a hole through the firewalls and NAT port translations on the network [JE03]. This, however, seems like a unnecessary problem to solve in this project.

Wireless 802.11b/g

Ruling out Bluetooth and 3G connections as a network for TileShifter, leaves the wireless 802.11b/g network, available in the HTC Desire. The 802.11b connection allows a theoretical throughput of up to 11 Mbit/s and the 802.11g allows a theoretical throughput of up to 54 Mbit/s, according to the IEEE standard on 802.11 [iee07]. This throughput should be sufficient for a near real-time network multiplayer game. Beyond the abilities of the HTC Desire, some other often seen wireless implemented standards are the 802.11a and the 802.11n, where 802.11a can deliver 54 Mbit/s in theory, and 802.11n goes to 150 Mbit/s in theory. The main difference between 802.11a and 802.11g is that 802.11a uses a higher frequency but has a shorter transmission range than 802.11g.

On a 802.11b/g connection, it is possible to control the routing and open ports on the device. Therefore it is easier to work with connections between the devices. It is possible to create an Ad-Hoc network which is connections directly between devices, without the use of routers and wireless switches. Larger Ad-Hoc networks for mobiles, have been subjected to a lot of research and are referred to as a MANET (Mobile Ad-Hoc Network), which is described in RFC 5444 [CDDA09]. Since the mobile devices are expected to move around, the connections need to change often. Every device in a MANET works as a router and forward traffic, it is therefore a complex task to keep the integrity of the routing in a MANET, this is done using NHDP (neighbourhood discovery protocol), which is described in RFC 6130 [CDD11]. To avoid making the project more complex than necessary, the MANET solution is not researched and it is assumed that all devices, playing TileShifter, are connected to a wireless network router using 802.11b/g.

One notable issue is, that if the game should be played over the internet, it is obligatory to create a NAT punchthrough solution, since most of the wireless network connections are behind a NAT address. This issue is not handled in this project, so a second assumption is, that the players of TileShifter are all on the same network.

2.3.2 Network Protocols & Architectures

The choice of hardware is the wireless 802.11b/g connection, the next step is to determine which protocols to use and investigate reasonable solutions for handling connections.

The following Sections discuss initially which protocol is viewed as the most reasonable, secondly the networking connection architecture is discussed and finally, a few existing solutions for game networking, are examined as possible

candidates for TileShifter.

UDP vs. TCP

Choice of protocol is important in connection with network multiplayer game development, especially when the game is supposed to feel like it updates in real-time on all the client devices. The reason is that there are many possibilities of protocols to choose from and because it has to be an optimal (in terms of speed and stability) protocol for the game to perform in the best possible way.

Choice of protocol becomes easier, when the expected type of data sent between devices during the game is known. The following description lists this expected data:

Connecting Joining a game session is not time critical. The important part of connection data is that it reaches the necessary parties.

Movement Essentially, the players need to receive updates on the whereabouts of others. This has to be sent often for a fluid gameplay experience.

Shots Whenever a player fires a shot, the other players need to get this information very fast after the event. Because there is an element of competition in the game, it must be delivered to all players implicated immediately.

Scoring Least important is the information on scores, defeats and victories. It is important in the context of game design, but in connection with timely delivery and reliability, it is less important.

Searching for a reasonable protocol for the purpose of multiplayer games is a time consuming task, various different, more or less custom and problem specific protocols exist, but very few turn up when looking for multiplayer game specific protocols. An assumption of why this is the case, is that game development seems rather new in the field of scientific studies and because most game developers are content with the use of either UDP (User Datagram Protocol), standardised in RFC 768 [Pos80], or TCP (Transmission Control Protocol), standardised in RFC 793 [Pos81]. It does however seem, that this trend is changing. Recent work, contributed in 2011, on a Power Aware Game Transport Protocol (PGTP) for Multi-Player Mobile Games [ASM+11] fuels the fire of the science of networking in games between mobile devices. Here Anand et. al. expect that the nu er of multiplayer games for mobile devices will increase in the near future, and address the important problem of battery

consumption by the networking hardware, while playing a game. Anand et. al. have provided interesting research, though for this project, the PGTP is viewed as too adolescent. This is mainly to avoid the risk of spending too much time on solving potential problems with an untested protocol.

Since a look into which protocols, the majority of other network multiplayer games use, reveals that it is common to use UDP and TCP, they are regarded as reasonable choices for a final implementation of TileShifter. The protocols are widely used, implemented as standard in most network programming libraries and are well documented. UDP performs well for timely deliveries of packages at the cost of not guaranteeing ordered delivery or any delivery at all [Pos80]. TCP is slow but ensures ordered delivery of all the data, as long as the connection is open, but requires the communicating peers to establish a connection prior to sending any packages [Pos81]. For TileShifter, this motivates using UDP packages for data transfers, because data needs timely delivery and since it is expected that the connections between peers often change, due to peers leaving and joining in the tile based Peer-2-Peer network.

Peer-2-Peer vs. Client-Server

Another question is how to organize the connected players in a network architecture. There are two major network architectures: a Client-Server solution or a Peer-2-Peer solution.

Related to games, the Client-Server architecture is often used. The reason being that it is easy to manage and synchronize the connected players, it is easy to replicate the server and there is a lot of control over the game sessions and the stability. The drawbacks of the Client-Server architecture is that it is not very scalable, since all players connect to a specific server. This means that the bandwidth of the server, is a bottleneck, limiting the number of players. Furthermore, the Client-Server architecture need a server available on the network, to allow the players to connect to.

Opposed to the Client-Server architecture, the Peer-2-Peer architecture does not need servers. Instead all players on the network communicate directly to each other. This can allow the game to scale, if the data is delivered between the connected players in an elegant manner. The drawbacks of Peer-2-Peer architectures are that there is little control over the game session, since every peer in the network has an opinion of what happens in the game, which often results in conflicting game states and desyncronisations. It is difficult to distribute the game updates to all of the connected peers. Peers should not be regarded as stable, meaning that the Peer-2-Peer network should be able to resolve problems such as failing, leaving and Byzantine peers (arbitrarily failing peers [CDK05]), but should also be able to handle churn (peer turnover).

A third opportunity is to combine the two architectures into a hybrid. The game concept of TileShifter was described in Section 2.1, is tanks driving around on tiles. This idea and the network architecture can be integrated in such a way that the separate tiles function in the Client-Server manner, while the tiles are interconnected in a Peer-2-Peer manner. A peer would be the master or server of a tile, and the peers having tanks on the tile would be clients of the master. Peers send their messages to the master, who distributes them to the other peers. Messages that need to be delivered outside of the tile can be sent to the neighbouring tiles. Notice that since all tiles are the same size and are square, a single tile can have up to four immediate neighbours. This way, the local messages and updates on a tile can function as a Client-Server structure, while the global messaging can be following the Peer-2-Peer architecture.

Using this hybrid model avoids conflicts on the tiles, while allowing the game a certain amount of scalability, provided every tank is not on the same tile at once. This is unlikely, since many tanks in the same place means large battles with many casualties. These dead players respawn on other tiles, distributing the players through combat. Avoiding the need for a dedicated server or tracker also means the game is playable at any time, and does not require the initial game starter to remain in the game.

Expanding on Peer-2-Peer

There are some research in the field of Peer-2-Peer in connection with games. One example of such research is an algorithm for large-scale, high-speed, Peer-2-Peer games, called Donnybrook [BDL+08]. The main idea of the Donnybrook Peer-2-Peer algorithm, is to introduce interest sets, calculated by each peer every frame. An interest set can be used to determine which peers have interest in what message updates. This brings the load on the bandwidth down. Bharambe et. al. use intricate formulae of distance and view direction to calculate the interest set. This concept of interest sets can be compared to the concept of tiles in TileShifter, as peers on one tile will be interested in updates from other peers on that tile.

Another example of the use of localization to determine who to send updates to, is formed in the paper "Peer-to-Peer Support for Massively Multiplayer Games" by Knutsson et. al. [KLXH04]. As the title of the paper suggests, the paper is minded on Massively Multiplayer Games, but the idea of location-based interest sets, failover resolution and use of replication, could be mapped to other game types as well.

The two papers' testing results indicate that the concept of location-based interest updates is a good solution to solving the bandwidth problem in large-

scale multiplayer games. This also goes for TileShifter. Instead of calculating distances and locations, however, the tiles form a natural boundary of interest of messages.

In the paper by Knutsson et. al. the use of a Dynamic Hash Table (DHT) has proven to be a good solution for managing and locating peers in a game session. The DHT is used in conjunction with Pastry, which is a widely discussed and tested scalable, decentralised object location and routing overlay for large-scale Peer-2-Peer systems, using a DHT [RD01]. Pastry is primarily an overlay network used for quick and easy localisation and fetching of specific files or data in a large distributed network, using a DHT. However, TileShifter does not require this capability, rather it is needed to distribute data from one peer to multiple other peers, as well as store highly mutable data, such as tank locations. For this reason a more specific overlay algorithm is created instead, but based on many of the same key concepts of the successful Pastry network overlay. For example, it uses neighbours as routing tables as well when sending global messages and maintains the network structure upon peer departure. The overlay algorithm for TileShifter is made for the concept of tile-based Peer-2-Peer network games, instead of file sharing, with mutable data instead of immutable data for which Pastry is designed. The tiles of TileShifter is a game design decision, that can be used as an optimisation for many parts of the game such as: unnecessary rendering on the limited resources of the Smartphone, aid in keeping connections between peers, and creating an exciting and dynamic map for the players. Dealing with Peer-2-Peer architectures, makes it impossible to assure connection between all peers at all times, especially under churn, because there is no single device or server that keeps an overview of the connected peers. So the task of the overlay is to repair the holes in the network and minimise the damage as much as possible, so that isolation of peers is kept at a minimum.

2.4 Choice of Game Engine

A good and robust game engine is advantageous when developing a game. The alternative of having to write native Smartphone device code is a very time consuming process and not the focus of this project. The game engine is expected to take care of the physics in the game, the graphics, the input and sound. Many game engines are available on the market, so the criteria to chose a game engine by are defined.

Smartphone support : Since the game is for Smartphones, the engine must have support for Smartphone development.

Network support : This being a network multiplayer game, the engine must support networking protocols, preferably UDP sockets.

Easy & fast to use : Development should be smooth and fast, to avoid waste of time on superfluous development issues with the engine, such as setup, configuration, learning and asset handling.

A very capable engine for this exact game development product, is the Unity3D engine [Uni11].

2.4.1 Using Unity3D

The Unity3D engine lives up to all the demands, provided the correct add-ons are acquired. It is very fast and intuitive to use for development of a game and there are a large community due to rising popularity and a free-to-use Indie version. Unity3D can be used to develop for several platforms using the same programming language (either JavaScript, Boo or C#). For this game the add-ons called Unity-Pro and Android-Pro are needed to make C# .Net sockets available in the game. The practical sides of Unity3D is that it is a WYSIWYG game editor allowing the developer to use standard C# .Net version 2.0 and libraries. Unity3D takes care of the translation to the native language on the chosen platform. Unity3D uses Nvidia PhysX as physic environment and has built-in sound, light and 2D-3D graphic rendering. This makes Unity3D ideal for this project.

Unity3D is used for creating the game, but since it provides such an easy development platform, it is also usable for creating testing and simulation applications. Using Unity3D to create network testing applications, ensure that the results are as close to the results of the game itself and due to the easy to manage 3D graphics, it is also very good for visual simulations.

Chapter 3

Design

The Design Chapter aim to provide a thorough depiction and description of various elements of the project. Initially, the Design Chapter will motivate viewing many of the networking problems as game design related choices instead, by explaining how game design can influence network functionalities and message passing.

An important part of the networking in the game, is the way the tiles are maintained as a map structure, and how this tile system is used to uphold connectivity between peers. The design of the algorithms managing this is a key part of the project and will be described here as well.

Besides the game itself network tests have been performed to find the throughput and bandwidth of wireless connections and a simulator has been created, to provide a test base of the tile algorithms mentioned. The design of these tests and simulations is described in this Chapter.

3.1 Game Design

The game design is chosen based on the consideration that it should feature relatively low complexity game mechanics, so it is easier to focus on the network specific concerns. The Peer-2-Peer and Client-Server hybrid leans itself to distributing the game to as many peers as possible, to allow the game to scale. Thus, the final game idea conceptualized is a player versus player game, where each player controls a tank. This tank is capable of driving around the playing area, as well as shooting directly ahead. The goal is to find, shoot and destroy opposing players to increase the player score and ultimately win the game, without getting hit and destroyed. Tanks will have a number of lives, and

once shot enough times eliminating all of a player's lives, the player will die and respawn on one of the tiles that he owns. Ownership in this case means the player is tilemaster of a tile, which is detailed further below. A score will be kept of how many other tanks a player kills, how many shots he has hit with and how many times he has died.

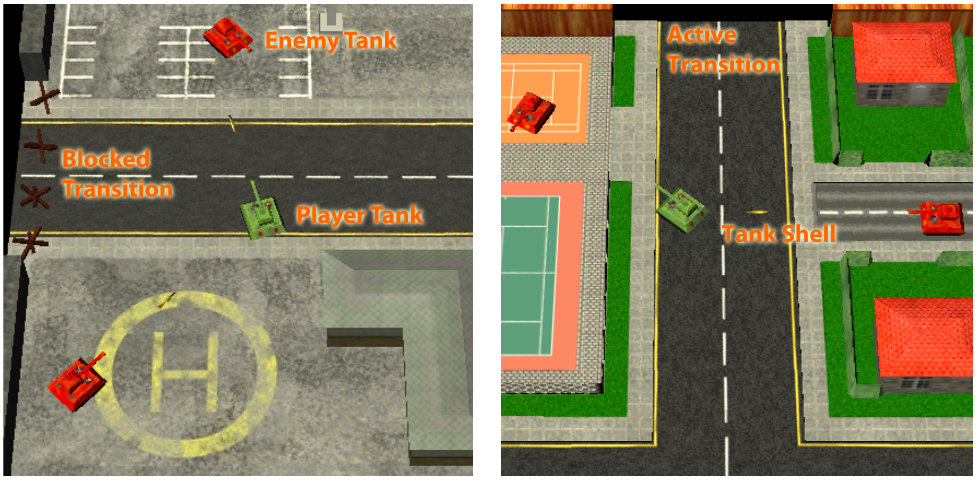
The game idea is simple, but allows itself to be expanded and modified easily with extra game rules, even rules made up by the players locally. This can include team based combat, zone control or even a drinking game. Many of these expansions, modifications and extra game rules are covered in the Section 5.3.2 on Further Development.

To allow the game concept of tanks to scale and fit the general idea of Peer-2-Peer, the game should be divided into parts. This is done using a tile system, where the playing area is made up of an amount of square tiles. Each tile represents a block of the town that is the battlefield, with four exits to adjacent tiles. Each tile is symmetric and thus different tiles can fit together in any combination, and there is no ambiguity as to where a player can enter and exit a tile. Tiles can still be different, however, using different tile sets, as long as the transitions are located in the same areas. Two different tile sets have been made so far, industrial and residential. A tile is created when a new player joins the game, and the joining player is assigned to be the tilemaster of this tile. This means as players join and leave the game, the game world grows and shrinks dynamically, and game worlds are unlikely to form the same way from game to game.

The game is controlled using the touch screen of the smart phone, with an analogue stick in the lower left of the screen, that directs the tank. The cannon is fired using the back button on the phone. In the game view of the phone it is only possible to see the tile that the player is currently on, which not only eases the computational load on the phone, but also lets the player focus on his active tile and not be distracted by other tiles. This also means that the game does not have to care what is happening on neighbouring tiles unless the player actually transitions to it. This ease the network load.

Screenshots of the implemented game, called "TileShifter", can be seen in Figure 3.1. In Screenshot 3.1a, set on an industrial tile, the player controlled tank is shown in green, and two enemy tanks in red. A blocked transition is also shown, meaning no tile neighbour exists in the western direction for this tile. In Screenshot 3.1b, the player is now on a residential tile, which shows a better view of a shell fired by a tank, as well as an active transition, meaning the player can go north to the tile's northern neighbour.

The game design and network fit together intentionally for this game, the tiles are managed in a Client-Server fashion by a tilemaster, who controls actions on the tile for which he is responsible. Each tank currently residing on this tile



(a) (b)

Figure 3.1: Screenshots from TileShifter.

acts as a client to the tilemaster. Each tile is interconnected in a Peer-2-Peer manner, where no one tile knows the entirety of the total system of tiles.

Tile

A tile represents a square part of the total map area arranged in a grid. A number of tiles fit together to form the total playing area for a certain game session. A tile has an ID, which is the IP address of the peer that owns the tile, with the periods removed. The IP address is unique for all peers on the network, keeping in mind that NAT is disregarded in this project. A tile includes a list of its neighbours, including the four immediate neighbours, but also neighbours an additional distance away, meaning a tile has twelve total neighbours in its neighbour list. This amount of neighbours is needed to facilitate updates and movements in the grid, because a tile needs to know of second degree neighbours to avoid conflicts from multiple tiles being interested in the same free coordinate. A tile also manages any tanks that are currently occupying it. A tile is depicted in Figure 3.2, where the blue tile is the peer, the immediate neighbours are marked green, and the red outline encircles the entire neighbourhood of the blue tile, containing twelve tiles. The peer responsible for a tile, is itself a tilemaster.

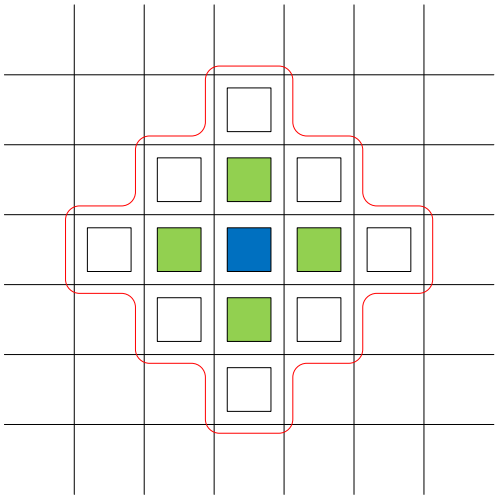


Figure 3.2: A Figure showing a tile and its neighbours.

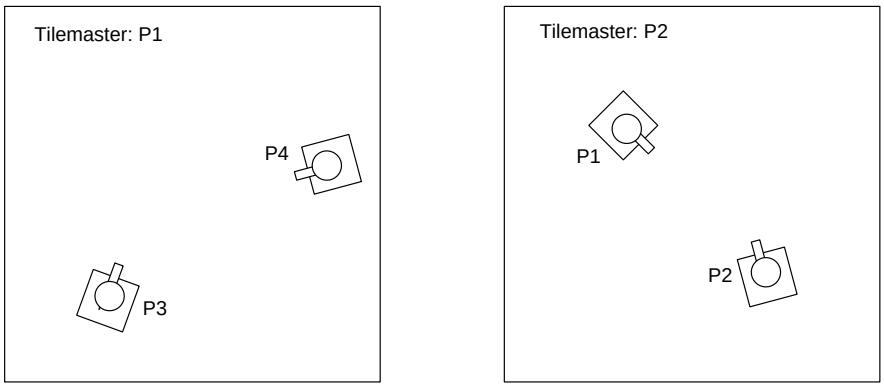
Tilemaster

The tilemaster is the peer that is responsible for a given tile. Being responsible for a tile means sending and receiving messages on behalf of the tile. It is therefore acting as a sort of server for the different peers using the tile. This covers tanks currently occupying the tile, as well as the tile's various neighbours. Tanks on the tile will send movement and shoot messages to the tilemaster, and the tilemaster will collect these messages to distribute them to all concerned peers. A tilemaster peer's tank does not necessarily have to be on the tile it is tilemaster for.

The concept of tilemasters is depicted in Figure 3.3. In this Figure, two tiles are shown, where player 1 and player 2 are tilemasters of a tile, however player 1 is not on his own tile. Figure 3.3c depicts the message flow that would result from this scenario: Player 1 sends tank data to player 2, and receives data back about player 2's tank. Meanwhile, player 3 and 4 send data to player 1, who collects their tank data and responds with the state of the tile to player 3 and 4.

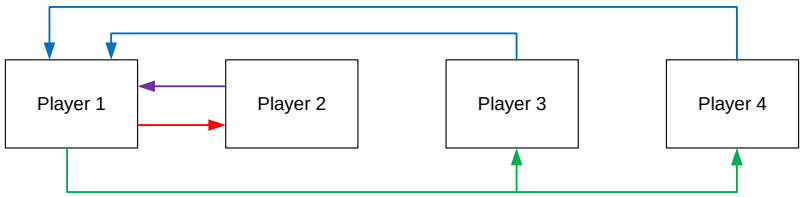
3.1.1 The Tilesystem & Grid

The system envisioned for the tiles and grid in the game, is advanced enough to warrant further details. To the tanks themselves — the avatars of the players — the tile system generates the playing field of the game. Behind the scenes, however, the tiles are much more. This Section explains the concepts behind the tilesystem, joins, leaves and updates, while the following Section 3.2 details



(a) Two tanks on player 1's tile

(b) Two tanks on player 2's tile



(c) Communication between peers

Figure 3.3: Figures showing the tilemaster concept

the algorithmic designs that conceptualise these ideas.

Abstractly speaking, the tiles are arranged in a grid, with each position in the grid corresponding to a coordinate in an (x, y) coordinate system, with the coordinate $(0, 0)$ at its center. Each tile is connected to its immediate neighbours in the grid, being able to send tanks, that drive onto the borders, onto the neighbouring tile in that direction, and vice versa. This means that a tank on the tile located at coordinate $(1, 2)$ in the grid is able to drive onto the neighbouring tiles occupying coordinates $(1, 1), (1, 3), (2, 2), (0, 2)$. If any of these grid coordinates are unoccupied, the passage in these directions will be blocked, in this case by impassable tank traps.

The grid is arranged such that tile placement gravitates towards the coordinate $(0, 0)$. This decision is based on the consideration that in order for players to enjoy the game the most and experience the most action, ideally they should be able to reach the tiles where other players are fighting within a

reasonable amount of time. The chosen gravity of $(0, 0)$ in the grid causes tiles to cluster together and not form long labyrinths of tiles. Furthermore, this help maximising the action in TileShifter, since the distance between players is reduced.

In addition, this clustering improves the robustness of the total tile map, such that players are still able to traverse from one end to the other without being isolated entirely because of some players and their associated tiles leaving the game. To further improve this, tiles will attempt to switch to coordinates closer to $(0, 0)$ when such an event occurs, so the tiles remain clustered. Tiles switching position happens behind the scenes, and players will only really know the tile they are on has moved if they move to a neighbouring tile. Though it has been decided to use the tile movement as a gameplay element, causing in-game earthquakes on the moving tile to notify players that the map has changed for them.

Closest to $(0, 0)$ is defined using Manhattan Distance [Bar11], and two tiles are compared using the following distance measurement, given tile A, with grid coordinates (x, y):

$$DistToZero\,(A) = |A.x| + |A.y| \tag{3.1}$$

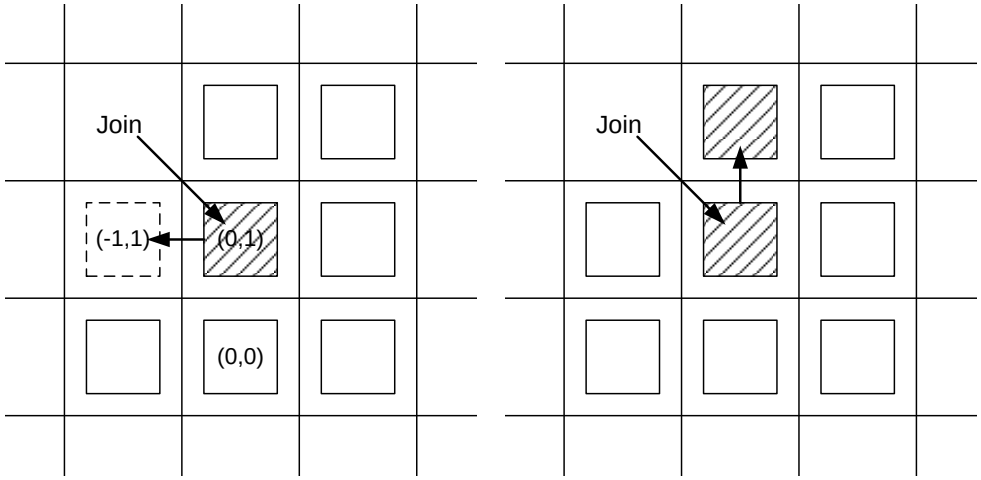
A tile is closer to $(0, 0)$ than another tile if the distance to zero is smaller than the other.

While not eliminating splitting problems altogether, this clustering allows the players to remain in the same game without being isolated and without having a dedicated server in a lot of situations. The robustness of this idea is further investigated and demonstrated in the simulation Section 5.2.

Joining

Each player joining the game session is assigned a tile of his own to be tilemaster for. This tile is placed on a free coordinate in the grid. This tile will be neighbour of the tile who's tilemaster peer he initiated the game connection to, in the ideal case. This joining case is depicted in Figure 3.4a, where a new peer joins the peer who is tilemaster of the tile in coordinate $(0, 1)$. This tile has a free neighbour spot in coordinate $(-1, 1)$, and the new peer is assigned this coordinate for his own tile.
A different case is when every immediate neighbour spot is occupied, the peer will be sent to one of these actual neighbours and attempt a join on this tilemaster instead. This continues until the peer finds a free spot for the tile. This case is shown in Figure 3.4b, here a new peer is trying to join the tile marked by the joining arrow, but all of the tiles' immediate neighbours are occupied.

The new peer is now told to try to join one of these neighbours instead, picked randomly. In this case of Figure 3.4b, the north neighbour is chosen.



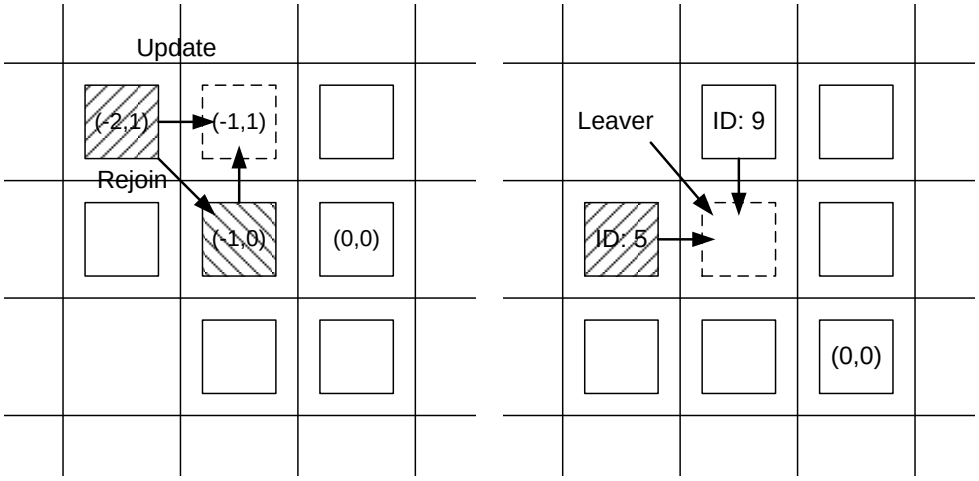
(a) Join (b) Indirect Join

Figure 3.4: Two different join cases

Leaving & Maintainance

The grid itself needs to be maintained in order for the criteria of clustering to hold adequately. Tiles can leave the game, or be positioned in conflicting coordinates on the grid. If a tilemaster leaves the game, that tile will be removed from the game, unless the tile is currently occupied by tanks other than the leaving tilemaster's tank, in which case the tile control will be transferred to one of its occupants.
If a peer leaves, crashes or otherwise disconnects from the game, then it leaves a gap in the grid. In order for these gaps not to create separate games and isolate players from each other, tiles further away from $(0, 0)$ than the leaving tile, whom are neighbours of this gap, can detect the gap and attempt to move to it. This will cause tiles to always be attempting to be as close to the center of the grid as they can be, causing a clustering. A small problem with using Manhattan Distance as the basis of this grid is, that given very specific join and tile placement scenarios, the tile map could form in long lines outward from $(0, 0)$, which defeats the purpose of the clustering. However, during the many simulations run in Section 5.2, this problem was never encountered.

Two such scenarios are depicted in Figure 3.5. The situation in Figure 3.5a could be caused by either the tile in coordinate $(-1, 1)$ leaving the game, or the tile in coordinate $(-2, 1)$ being placed this way by a join. The tile in



(a) Update and move

(b) Two nodes interested in the same coordinate

Figure 3.5: Two scenarios caused by a node leaving

coordinate $(-2, 1)$ discover the open coordinate in $(-1, 1)$, which is closer to $(0, 0)$, and looks in its neighbour list to asks neighbour $(-1, 0)$ to move it to the open coordinate, leaving the old coordinate $(-2, 1)$ vacant.
In the other situation, depicted in Figure 3.5b, a tile has left the game in the center of a group of tiles. Both the tiles with id 5 and id 9 see this closer coordinate open and seek to move there. First, however, they search their neighbour list for any known neighbours that could also be interested in the coordinate, discovering each other. This conflict is resolved by the lowest id getting priority, and as such id 5 moves to the open spot while id 9 remains where it is.

Game design wise, this movement of tiles can even aid the gameplay, such that any time a tile is moved and repositioned in the grid, the players on this tile will experience an earthquake or a similar event in the game, to signify that the tile they are on has switched to a new position in the game world. This should give the players on a tile that has just moved a sense of exploration, to discover if a battle is occurring on the new neighbour tiles that they can take part of.

3.2 Tile Algorithms

A central part of the game, both in terms of design and network structure, is the tile maintenance algorithms. The tile system is a result of many con-

siderations of how the peers connect to each other and how the in-game map are going to be structured as new peers join and other peers leave or disconnect. These are the same concepts as explained in Section 3.1 above, but in this Section, a more algorithmic approach is taken with concepts of joining, maintenance and updates. The algorithms as a whole are a relatively complex system, which is why this section aims to provide a better understanding of the algorithms including argumentation for the choices made in the algorithm design. The following requirements have been specified for the tile algorithms to ensure that they perform satisfactory:

Robustness Any player can join or leave the game at any time without inconveniencing the other players to a great degree.

Reachability It should be possible to reach other tiles in the game, disregarding a possible split. Tiles should be placed in such a way that there is a short path to the point of battle.

Scalability The game should scale to a certain amount of players, relevant in cases such as: A full auditorium, a bus, a train or an aircraft or other usecases.

Decentralised No dedicated server or tracker is required for the game to run.

The algorithms have been split into peer actions; Join, update and keepalive, and the descriptions of these parts follow the normal life cycle of a peer joining a game session.

3.2.1 Variables, Intervals & Timeouts

Several timeouts and intervals are used in the following algorithms, it is helpful to know how these are defined and what their overall purpose is. The description below provides a quick overview of these, the values provided are in seconds and they are the values used in the simulator. The value numbers may be different in TileShifter, but the relation between the values are the same for simulator and game.

COORDTIMEOUT $= 5$
— Time that should pass before a neighbour is deemed as timed-out and therefore considered disconnected or crashed.

COORDUPDATETIMER $=$ **COORDTIMEOUT** $\times 3$
— Time that should pass before an update runs. Used to avoid updates each frame to save processing power.

INTERVALKEEPALIVE $= 2$
— Time that should pass between sending each keepalive message. To avoid sending keepalive messages each frame, saves the network bandwidth.

LONELYTIMER $= 20$
— Time that should pass before a check, if the peer is isolated and should try to rejoin to a peer closer to the coordinate $(0, 0)$.

3.2.2 Joining

The initial step of a peer is either to join a running game session or create a new session. Creating a new session does not require any algorithmic measures, the device has to initialise the first tile in the game, and the player controlled tank. Joining a session, however, is done by creating a connection to another peer who has created a game session already. Since the game is a server-less Peer-2-Peer game, the joining peer needs the IP address of another player in the running session.

The join is written as Code Example 3.1.

```
1  Join(IP)
2      Ask known peer to join by ip
3      Known peer looks for a free coord using FindNextFreeCoord
           ()
4      If a free coord is found
5        Fetch IPs of the neighbours relevant to the joiner, at
             the new coord using GetRelevantNeighbours()
6        Add the joining peer to the list of neighbours at the
             coord
7        Send the new coord and the relevant neighbours to the
             joining peer
8        Place tile at the free coord
9        Initialize() the joining peer
10     Else
11       Send the IP of a random(Could result in livelock)
             neighbour peer, using FindNextFreeCoord()
12       Run Join() on recieved IP
```

Code Example 3.1: Pseudocode of the *Join*() algorithm.

The idea behind the *Join*() algorithm in Code Example 3.1 is that when a new peer joins an existing peer, the peer needs to check if it has a free spot (coordinate) either directly to the north, east, south or west. If a free spot is found, the coordinate and the known relevant neighbours for that coordinate (which is found using the *GetRelevantNeighbours*() in Code Example 3.3) is sent in reply, and the joining peer is now initialised as being placed in the grid and having joined the game properly. This case was depicted in Figure

3.4a. If none of these spots are free, it should instead inform the joining peer that it must join on one of the neighbours. The *Join*() algorithm uses the *FindNextFreeNode*() algorithm to check if a neighbouring spot is free and to find a random neighbouring peer. This case can be seen in Figure 3.4b. The *FindNextFreeNode*() algorithm used to discover free coordinates or pick a neighbour to join is seen in Code Example 3.2. The livelock has solutions in Section 5.3.1 about further development, if it becomes a problem.

```
1 FindNextFreeCoord()
2   Find the neighbour coord closest to zero (immediate north,
        east, south, west) among free coords.
3   If one to more free coords are found
4     return a random free coord
5   Else
6     Return a random neighbour
```

Code Example 3.2: Pseudocode of the *FindNextFreeNode*() algorithm.

The *FindNextFreeNode*() algorithm in Code Example 3.2 looks through the list of neighbours and finds the free coordinate closest (Manhattan Distance) to coordinate $(0, 0)$, if there is one at all. If there are no free coordinate, then return a random neighbouring peer's IP to join on instead.

```
1 GetRelevantNeighbours(coord)
2   Foreach existing neighbour AND the coord
3     Calc the absolute value of the neighbour coord - coord to
          check
4     If the value is equal to 2 or less (manhattan distance)
5       Add the neighbor to a return list
```

Code Example 3.3: Pseudocode of the *GetRelevantNeighbours*() algorithm.

Since the peer on which a new peer joins likely knows some of the neighbours to the coordinate the joining peer is placed on, it should inform the joining peer about the neighbours relevant to that coordinate. *GetRelevantNeighbours*() in Code Example 3.3 finds these neighbours given a specific coordinate. It does so by subtracting the coordinate of each neighbour from the coordinate to check. If the neighbour coordinate is within two steps of Manhattan Distance of the coordinate, the neighbour is deemed relevant, and is returned. This list of relevant neighbours is then forwarded to the joining peer along with the join message. When the peer has been initialised and placed it is time to start "Updating".

3.2.3 Updating

Once a peer have joined the game session, received a tile coordinate and been placed in the grid, it is time to start performing updates and maintenance

tasks on the grid. These tasks include keeping neighbour lists up to date, sending keepalives, moving around the grid and resolving conflicts. Since it is unreasonable to expect graceful leaves and changes in a Peer-2-Peer network, it is not possible to foresee when maintenance tasks should be performed. Performing the tasks at given intervals is a solution to this.

Using Unity3D for the development, enforce the use of the built-in *Update*() method. This method is executed once per frame by the Unity3D engine. This makes it ideal for runtime maintenance of the peers and coordinates. Notice that since it is executed before every single frame, it is best to avoid high complexity in the algorithms. Furthermore, it is not necessary to perform the maintenance tasks each frame, so timers are used to set relevant intervals for these tasks.

See the *Update*() in Code Example 3.4.

```
1  Update()
2    If peer is initialized
3      ...
4      If peer is not coord 0,0
5        Decrease LONELYTIMER
6        If LONELYTIMER is below 0
7          FindNeighbourClosestToZero()
8          If the current peer is closest to zero
9            Join() on either initially joined peer or a random/
                 closestToZero peer known from the global score
                 messages
10         Else
11           Reset LONELYTIMER
```

Code Example 3.4: Pseudocode of the *Update*() algorithm.

The *Update*() algorithm in Code Example 3.4 is executed when the peer has been initialised. Its main purpose is to make sure the *UpdateNode*() and the *SendKeepAlive*() algorithms are being executed in reasonable intervals. In addition it removes timed out neighbours from the neighbour list. These are simple timers and have been omitted from the Code Example. The intervals for these timers were defined in Section 3.2.1. Furthermore, it checks if the peer has become separated from a larger game. The check is performed if the peer is not placed in the coordinate $(0, 0)$. This check works, since the concept of the tiles is, that they cluster up around the coordinate $(0, 0)$ throughout the games session's lifetime. If the tile is detected to be isolated, because no neighbours are closer to $(0, 0)$ than itself, it will attempt to rejoin the game by joining a peer it have discovered through global messages containing scores and other information. This information might not be current, and it is possible for the tile to be unable to find another tile, in which case it will stay where it is. If this should happen, the peer have not been participating in the game

very long, and drives on its own tile. An example of a tile being isolated is depicted in Figure 3.6.

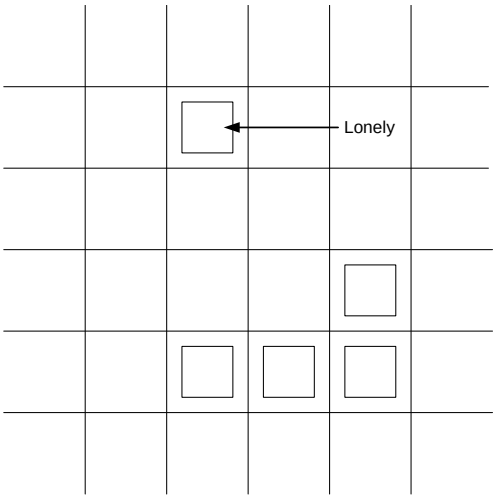


Figure 3.6: A isolated tile

The *UpdateNode*() algorithm, seen in Code Example 3.5, has the purpose of discovering empty grid coordinates and moving the tile closer to the coordinate $(0, 0)$ if possible.

```
1  UpdateCoord()
2    Foreach neighbour that does not exist
3      If the empty neighbour spot/coord is closer to zero than
           the peer itself, AND is not more than 1 manhattan
           distance away
4        Look for a neighbour that exists, which is closer to
             zero than the peer itself
5        If a closer to zero peer is found
6          Join() the neighbour closer to zero
7        Else
8          Check all neighbours to see if the peer is alone
9          If we are not alone
10           check neighbours who could be a candidate to move
                 closer to zero coord based on coord position
11           If the peer has the lowest Id/ip nr amongst
                 candidate neighbours
12             allow peer to move to the spot closer to zero
13           Else
14             // Do nothing, let the other peer move (conflict
                   resolved)
15         Else
16           // Do nothing. Peer could be lonely, a later update
                 resolves this issue
```

Code Example 3.5: Pseudocode of the *UpdateNode*() algorithm.

UpdateNode() in Code Example 3.5 checks each of the peer's neighbours, to see if a neighbour position in a coordinate closer to $(0, 0)$ is available, maybe because of a disconnection or having joined in a coordinate with free neighbours. If the peer knows a neighbour closer to $(0, 0)$ than itself, it will *Join*() that particular peer to get a new coordinate and move to the open coordinate. This rejoin still preserves any neighbours that are relevant to the new coordinate. The scenario was depicted in Figure 3.5a.
If the peer is the closest to $(0, 0)$ among its neighbours, it will check if there are any other neighbours with the same Manhattan Distance to the free coordinate as itself. In this case this neighbour is also a candidate for the new empty coordinate. There might be two or more candidate tiles that are able to move to the same open coordinate, and to avoid race conditions and further conflicts on this account, a general rule is applied stating, that the peer with the lowest IP/ID value takes highest precedence. The tile with the lowest value allows itself to move to the open coordinate, and the other candidates with higher values stay at their current positions. This scenario was depicted in Figure 3.5b, where the tiles with ID 5 and 9 are candidates to move to the same open coordinate, and the tile with ID 5 is allowed to move.

3.2.4 Keepalives

Keepalives are important, since they are used to uphold the connectivity and discover disconnected neighbours. Furthermore, they are used to maintain and update neighbour lists of each tile. The keepalive messages in TileShifter, are global messages, that hold scores and are propagated to each neighbour. Keepalive messages are sent every time the interval for keepalive has passed, these are sent to every peer in a given peer's neighbour list. Neighbours from which no keepalives have been received for the duration of the timeout, are considered disconnected from the game. Keepalives are intended to work as global messages for the game, which includes kills, scores and other values that peers on other tiles are interested in. It is intended that they should piggyback as much additional information as possible, to avoid unnecessary overhead from extra package headers, and wasting bandwidth.

The *RecieveKeepAlive*() in Code Example 3.6 is more complex. First of all, it handles adding newly discovered neighbours to the neighbour list of the tile, as well as resetting the timeouts for already existing tiles. It must be able to act on conflicting neighbours, which can be problematic, since all peers have their own interpretation of what is the correct game state.

```
1 RecieveKeepAlive(Id/Ip, coord, relevantNeighbours)
2   If neighbour sending keep alive is still relevant
3     If peer already has a neighbour registered to the spot/
          coord
```

```
4       If the peer is the same
5         Reset the COORDTIMEOUT for the neighbouring peer
6         Foreach relevant neighbour received
7           If we have no peer registered at the position AND it
                is relevant to the peers interest
8             Add the relevant neighbour to the list of
                  neighbours
9           Else
10            // Silently discard the keepalive - the peer is
                  not of our interest.
11            // We might have a conflict, but since it is just
                  a relevant neighbour from the neighbour, it is
                  not sure we can trust it.
12      Else
13        // We have a conflict
14        If no investigations are pending for the neighbour
              coord
15          Start a ConflictInvestigator() on the neighbour
16    Else
17      Add the keep alive sender peer, to the peers list of
            neighbours at its coord
18      Foreach relevant neighbour the recieved
19        If we have no peer registered at the position AND it
              is relevant to the peers interest
20          Add the relevant neighbour to the list of neighbours
21        Else
22          // Silently discard the keepalive - the peer is not
                of our interest.
23          // We might have a conflict, but since it is just a
                relevant neighbour from the neighbour, it is not
                sure we can trust it
```

Code Example 3.6: Pseudocode of the *RecieveKeepAlive*() algorithm.

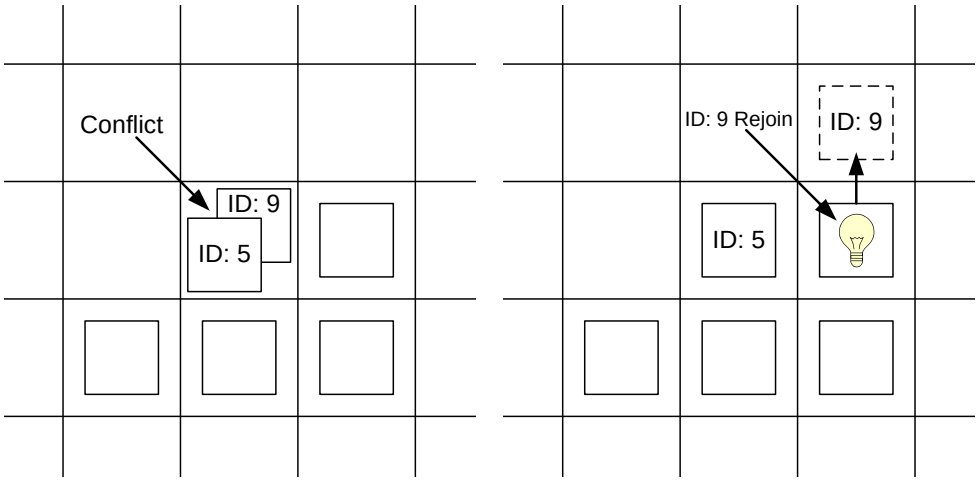
First part of the *RecieveKeepAlive*() in Code Example 3.6 checks the coordinate of the peer, from which the keepalive arrived, to make sure that the message is still relevant (the receiver might have moved since the last message). If the coordinate is relevant, the ID of the message is compared with the ID of the tile, registered in the neighbour coordinate, to check if there is a conflict. A conflict occurs if the ID of the peer registered in that coordinate, is different from the ID of the received keepalive message's peer.
If a conflict is found and it is not already running for that particular conflict, the *ConflictInvestigator*() in Code Example 3.7 is invoked. If, however, the peer is found not to be in conflict, or if the neighbour position is empty, then the peer is added and/or timeout is reset. In any case, the neighbours sent with the keepalive message are examined in the same way, however, this is merely to fill empty spots in the neighbour list. Neighbours' neighbours are not investigated in *ConflictInvestigator*(), because it is not sure the information received about them from the neighbour, can be trusted. It is not sure that they are still relevant and alive. A keepalive message from the neighbours'

neighbours should also arrive shortly, provided they are still alive connected.

```
1 ConflictInvestigator(neighbour)
2   Wait for INTERVALKEEPALIVE + COORDTIMEOUT // maybe the
        conflict resolves itself
3   If the neighbour spot/coord still holds a neighbour AND the
        neighbour the peer knows to be in the spot/coord, is not
        the one known to be in the spot
4     Ask neighbour with the highest ID/IP nr in the conflicting
          position to Join() again
5     Update the neighbour list accordingly
6   Else
7     // The conflict was resolved peacefully, stop the
          investigation.
```

Code Example 3.7: Pseudocode of the *ConflictInvestigator*() algorithm.

The *ConflictInvestigator*() in Code Example 3.7 is responsible for figuring out how to solve a conflict. Initially the investigator waits for enough time for a peer to time out, this is because the conflict might be due to another neighbour knowing, that the old peer on the coordinate was disconnected, and have placed a new peer in the empty coordinate without the current peer having timed the old peer in that location out yet. If the conflict still exists then the peer with the highest IP/ID is instructed to rejoin on the tile that discovered the conflict. An example of one such conflict is depicted in Figure 3.7.



(a) Conflict between tiles 5 and 9 (b) Conflict resolved

Figure 3.7: Conflict scenario

In this example, the first part in Figure 3.7a shows two tiles, ID 5 and ID 9, being placed in the same grid coordinate. This could be caused by both

tiles joining at the same time to different tiles in range of the same tile. In Figure 3.7b this conflict has been detected by the peer with the tile signified by the light bulb, the conflict is then investigated. After investigation, tile ID 5 is allowed to stay, due to lower ID, while ID 9 is instructed to rejoin and is assigned a new coordinate.

3.2.5 Disconnecting

Disconnection from the game is handled by timeout. Therefore no algorithms specifically for leaving, have been created. This is chosen from the deliberation that the game should be able to handle random failures and disconnects without any graceful leaving mechanism. The reason for this being that when a game is played on a Smartphone, leaving a game is seldom graceful. Take for example the situation where the phone receives an incoming call or when the user needs to do something else. In both cases the game would just lose focus on the phone, not run any cycles, and be disconnected. The user may also forcibly close the game, also causing a disconnect.

All of these algorithms work together to create the tilesystem and manage the grid without a dedicated server, the system is entirely self organized. Players can join and leave on a whim, to any peer in the system, and be included in the game. These peers will attempt to keep the grid in a non conflicting state, and keep the tiles clustered around $(0, 0)$. This tile system is deeply connected with the gameplay, as tile movement and the tiles themselves are building blocks of gameplay.

3.3 Test & Simulation Design

In this project, two different tests or simulations were performed. The first test's purpose was to ascertain the capabilities of the Android Smartphones used to test the game. This was done to examine the packet sizes, optimal to use for transferring wireless data, as well as how the effect of different wireless networks affected the round trip time.

This test was designed using two applications, one being the sending application and the other being the receiving application. These two applications were built to a Smartphone each, using Unity3D, and the Smartphones were connected to the same wireless network. The sending application will transmit a number of packets of a fixed size, marked with timestamps and sequence numbers, to the receiving application on the other Smartphone. The receiving application will immediately send the packet unmodified back to the sending application, which will measure the resulting round trip time.

These results is logged, and once all the packets of a test set have been sent and received. A mean round trip time could be calculated, for each different packet size. The results of this test can be seen in Section 5.1.

A more detailed simulation is performed as well. This simulation has as its goal, to determine whether the algorithm designed for handling the tile system and the grid, is robust enough to live up to the requirements set forth in the Problem Statement 1.1 and the Algorithm Design in Section 3.2. This simulation is executed locally, using a simulator designed for the job in Unity3D, without using a Smartphone. The main reason for not using Smartphones for the simulations is the difficulty of acquiring the number of Smartphones required to adequately test the real game. The simulator will allow execution of many tests overnight, stressing the algorithm with a series of joins and leaves.

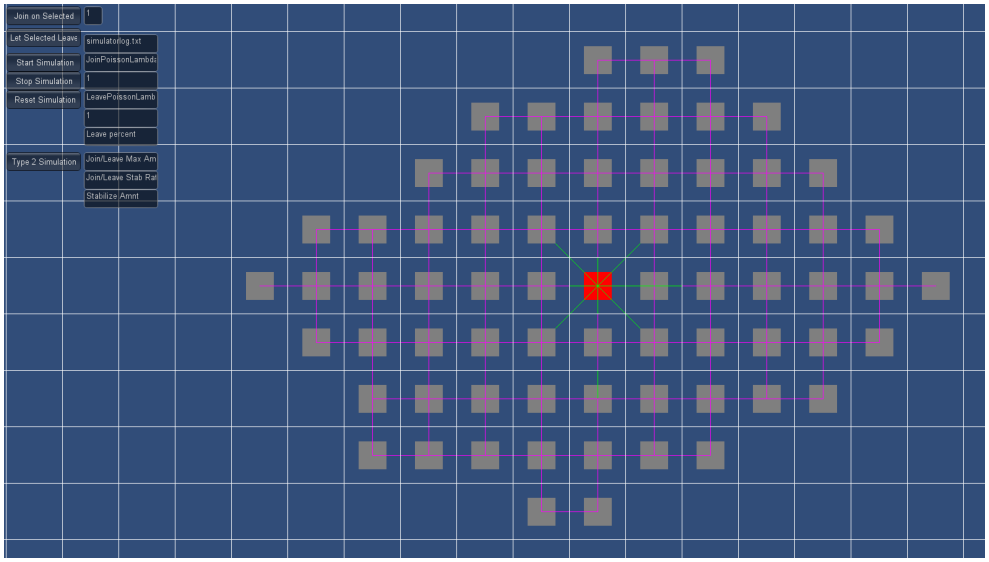


Figure 3.8: A screenshot showing the simulator in action.

This simulation is designed to have a single program running, with a single simulator. This simulator is able to instantiate any amount of peers to join the system, it can also make any amount of peers leave the system again. In order to allow the simulation to run unattended, and deal with randomly selected input, at random times, the simulator is extended with support for Poisson Processes, which are used to select average wait times between each join or leave event. This is based on the rate or intensity of the Poisson Process [Wei11, Hen09]. This allows a setup of the simulation with certain values for join and leave rates, and if required, a certain percentage of current peers, as the leave ratio.
The simulator is also extended with support for simulations, that will stabilize

at a certain number of peers in the system as the population mean, in order to test more realistic scenarios of game usage.

In Figure 3.8, a screenshot of the simulator is shown. The simulator can be seen with a number of active tiles currently in the game, the pink lines showing immediate neighbour connections. The red tile is the selected tile, in case a scenario requires joins or leaves on a specific tile. Green lines show every neighbour of the tile. In this case, the tile marked is $(0, 0)$. The results of the simulations and the conclusions about them, can be seen in Section 5.2.

Chapter 4

Implementation

The Implementation Chapter highlights important and interesting implementation choices made throughout the development phase. The implementation examples and explanations are for both the game and the simulator. The aim is to further the understanding of difficult choices made during the game and simulator implementation.

4.1 Game Implementation

This Section of the Implementation Chapter explains noteworthy examples of the programming in the developed game. The main part of the explanations are based on, or related to, the networking aspects, since networking is the main focus of this project.

Code examples and explanations are divided based on functionality, in the following Sections.

4.1.1 Sending & Receiving Packages

Sending and receiving packages are done using the C# .Net Sockets, more specifically using the UdpClient class. To ease the interfacing between the game and the UdpClient, a UdpSender and a UdpReceiver class have been implemented. UdpSender is a simple class that initialises and stores a UdpClient that can be used for sending package data, while the UdpReceiver is a bit more complicated. The peer should be listening for packages with game updates at all times and this has to be done without interrupting the game, therefore the UdpReceiver runs as a thread behind the game. As opposed to multiple UdpSenders, there is only a single UdpReceiver per peer since the

UdpReceiver is listening on all data arriving on a specific port. The game needs an UdpSender for each peer it needs to send data to. Unity3D does not support threading well, so a great deal of caution has to be taken when using and stopping the threads. *StartReceiverThread*() creates a UdpClient bound to a provided port and starts the listening thread, it is seen in Code Example 4.1.

```
28      public void StartReceiverThread(int port)
29      {
30          mUdpReceiver = new UdpClient(port);
31          if (!mThreadRunning)
32          {
33              mUdpReceiverThread.Start();
34              mThreadRunning = true;
35          }
36          mThreadStarted = true;
37      }
```

Code Example 4.1: The *StartReceiverThread*() method. Initialising UdpClient and starting listener thread.

The thread started in the *StartReceiverThread*(), which can be seen in Code Example 4.2, runs as long as the game runs and fires the event *DataReceived*() as soon as data has been received.

```
56      private void UdpReceive()
57      {
58          while (true)
59          {
60              if (!mThreadStarted)
61                  break;

63              mReceivedData = mUdpReceiver.Receive(ref mIPEP);
64              if (DataReceived != null)
65                  DataReceived(this, EventArgs.Empty);
66          }
67      }
```

Code Example 4.2: The *UdpReceive*() method. Thread that listens for data on the created UdpClient.

The event saves the received data into a byte array, before feeding it to the data decoder, which is described in Section 4.1.2. The resulting decoded data packages are inserted into a package queue, read by the game. The byte array and the package queue (described in Section 4.1.3) works as a double buffering of the received data packages, by inserting them at one end, and removing them from the other. Only packages in the queue at the time the *Update*() method is invoked are processed, adding later arriving packages to the queue for the next processing.

4.1.2 Encoding & Decoding Packages

Sending updates between peers is done as arrays of bytes, but this needs to be converted to a format that is understandable by the game. For this, two different methods were tested, one of which was to create a serialisable object, that can be converted to a byte array by C#. The other was to create the byte array manually from a string formed from a predefined syntax. The second solution of creating the byte array manually, proved to be more efficient in connection with the size of the package sent on the network, because there is a lot more control over what the package actually consists of. This means that the size of the packages can be reduced, as opposed to the solution of the serialised package.

The *Encode*() and *Decode*() methods convert between a byte array sent or received and a *PackageData* object, that can be read in the game. The *PackageData* is seen in Code Example 4.3.

```
 8 public struct PackageData
 9 {
10     public PackageHandler.packageType type;
11     public string ip;
12     public PackageHandler.subPackageType subType;
13     public Vector2 pos;
14     public Quaternion rot;
15     public int misc;
16     public List<Tile.tileNeighbour> neighbours;
17     public List<TankRepresentation> tanks;
18 }
```

Code Example 4.3: The *PackageData* structure. Denoting an object called *PackageData*

Notice in Code Example 4.3 that a *Vector*2 and a *Quaternion* are Unity3D defined object types.

Since the messages sent between peers are created manually, they can be constructed as a string with the needed information for a given message. Not all information in a *PackageData* object is required for every message type, so to save bandwidth, only the required information for the specific message type should be inserted into the message. It is important to define a strict syntax for the messages. It begins by defining an integer that denotes the type of message, followed by an integer indicating the subtype of the message. When both the sending and receiving peer know exactly how the message is constructed, it is possible to concatenate strings with raw information delimited with markers. As example of this way to create the messages see Example 4.1. The example is a reference of a *ConnectReply* package type, with subtype *SpawnHere*.

$$[TYPE]\#[SUBTYPE]\#[COORDX],[COORDY]\#[TILETYPE]\#\underbrace{[IP],[COORDX],[COORDY]\$}_{ForeachNeighbour}$$

$$3\#2\#0,0\#1\#10.10.10.1,1,0\$\cdots \tag{4.1}$$

In the *SpawnHere* Example (4.1) the first integer (3), denotes that the message type is a *ConnectReply* message. There are a few different *ConnectReply* messages, so the second integer (2) denotes that it is a *SpawnHere* reply. This message is designed to inform a connecting peer, that it can create the tile of a specific type (type 1 in the example) on the device, with the specified coordinate ($(0,0)$ in the example). Notice that enemy tanks on the current tile will be created when receiving the first movement update from the tilemaster peer. The last part of the message is the neighbours of the tile $(10.10.10.1,1,0)$. These are provided for two reasons, one is that the peer spawning on the tile, can be used as a replicating peer if the original tilemaster disconnects, but also to indicate what directions the peer can drive to leave the tile.

4.1.3 Dequeuing Packages

Recall that the received packages are inserted into a queue of updates. This queue is emptied every *Update*(), which is executed before each new rendered frame. In Code Example 4.4, a fragment of this *Update*() method is seen. This part of the method, takes a snapshot of the queue and acts on the queued events.

```
1  int queueSize = mNetEventQueue.Count;

3  for (int i = 0; i < queueSize; i++)
4  {
5    queuedEvent qe = mNetEventQueue.Dequeue();
6    switch (qe.PackageData.type)
7    {
8      case PackageHandler.packageType.Connect:
9        packetReceivedConnect(qe);
10       break;
11     case PackageHandler.packageType.Move:
12       packetReceivedMove(qe);
13       break;
14     case PackageHandler.packageType.Shoot:
15       packetReceivedShoot(qe);
16       break;
17     case PackageHandler.packageType.ConnectReply:
18       packetReceivedConnectReply(qe);
19       break;
```

```
20      case PackageHandler.packageType.MoveTile:
21        packetReceivedMoveTile(qe);
22        break;
23      default:
24        Debug.LogWarning("PackageType not implemented!");
25        break;
26    }
27  }
```

Code Example 4.4: Part of the *Update*() method that acts on package type.

The package receiver is running as a background thread, feeding into this queue, so it could result in problems if the receiver thread and this *Update*() method writes and reads the same elements in the queue, at the same time. Line 1 in Code Example 4.4 stores the length of the queue at the beginning of the execution, so that the following *for*-loop dequeues only the enqueued events and not events inserted while the *for*-loop is running. Any events enqueued that were not included for this particular *Update*() execution, will wait till next frame.

4.1.4 Act on Messages, Example

The *packetReceived*...() methods perform the needed actions for the received messages. As an example, Code Example 4.5 depicts how a shoot message is handled, using *packetReceivedShoot*(). Notice that the method is not executed when the player fires a shot, but only when the player receives a shoot message from another player, or from the tilemaster.

```
1  private void packetReceivedShoot(queuedEvent qe)
2  {
3    // Test if the current tile is the tile of the shot
4    if (mPlayerScript.currentPlayerTile.TileCoord.Equals(new
         Tile.tileCoordinate((int)qe.PackageData.pos.x, (int)qe.
         PackageData.pos.y)))
5    {
6      GameObject tankObject;
7      if (mPlayerScript.currentPlayerTile.tanks.TryGetValue(qe.
           PackageData.ip, out tankObject))
8        tankObject.GetComponent<Cannon>().Shoot();
9    }

11   GameObject masterTileObject;
12   if (mPlayerScript.masterTiles.TryGetValue(new Tile.
         tileCoordinate((int)qe.PackageData.pos.x, (int)qe.
         PackageData.pos.y), out masterTileObject))
13   {
14     byte[] shooter = PackageHandler.Encode(qe.PackageData);
15     foreach (string p in mPlayerScript.currentPlayerTile.tanks
           .Keys)
```

```
16      {
17        // Dont send the package back to the sender, and dont
              send it to ourselves..
18        if (p.Equals(qe.PackageData.ip) || p.Equals(Statics.
              playerIp))
19          continue;

21        if (!peers.ContainsKey(p))
22        {
23          UdpSender localSender = new UdpSender(p,Statics.PORT);
24          peers.Add(p, localSender);
25        }
26        peers[p].SendPackage(shooter);
27      }
28    }
29  }
```

Code Example 4.5: Part of the *packetReceivedShoot*() method that acts on a received shoot package type.

There are three different situations or states for the peer that receives the shoot message:

- The peer is a tilemaster, and must send the shoot to all peers moving on the tile, but the tilemaster is not on the tile, so the shot should not be displayed for himself.
- The peer is a tilemaster, and must send the shoot to all peers moving on the tile, and the tilemaster is on the tile, so the shot needs to be displayed on his current tile as well.
- The shoot comes from a tilemaster, and the receiving peer is not a tilemaster on the current tile, which means that the shot needs to be displayed on the current tile.

Lines $4-9$ in Code Example 4.5 test if the shot is fired on the current tile. If that is the case, the shot is initialized and fired on the player's screen.

Lines $12-28$ in Code Example 4.5 test if the peer receiving the shoot message is the tilemaster of the current tile. If so, it is obligated to send the shoot message to all peers on the tile, except itself and the peer that initially sent the shoot message.

4.2 Algorithm Implementation

During the implementation of the algorithms, mainly when creating the simulator, a few changes to the original idea were implemented to improve the tile

algorithms. In this Section significant implementation examples are brought forward as code examples and explanations. Notice that the code examples are mostly from the implementation of the simulator, since the algorithm is more apparent and not entangled in the game implementation.

4.2.1 Coordinate Closest to Zero

The first example is based on determining which tile is closer to $(0, 0)$ than another. This is a widely used method that can be found for neighbour objects, tile objects or tile coordinate objects by overloaded methods. In Code Example 4.6 the $findCoordClosestToZero()$ takes two $TileCoord$ objects which is a structure with two integers, x and y, that imply a coordinate.

```
1  private TileCoord findCoordClosestToZero(TileCoord a,
       TileCoord b)
2  {
3    int aPos = Mathf.Abs(a.x) + Mathf.Abs(a.y);
4    int bPos = Mathf.Abs(b.x) + Mathf.Abs(b.y);

6    return bPos < aPos ? b : a;
7  }
```

Code Example 4.6: $findCoordClosestToZero()$ return the coordinate closest to $(0, 0)$.

In Code Example 4.6, the $findCoordClosestToZero()$ uses Manhattan Distance as was described in Section 3.1.1, to determine whether a or b is closer to $(0, 0)$ by adding x and y for a and b individually. It could occur that the two values are of an equal value, in which case a is returned. This choice means that a is favoured, which has to be taken into consideration before executing the method, as the first parameter has to be chosen with care.

4.2.2 Dealing With Joining Peers

When a new peer, b, joins the game session, on a given peer, a. a must check if there is a free coordinate next to the tile of peer a. Since peer a has knowledge of neighbours of up to two steps of Manhattan Distance. This could be a neighbour's neighbour, but when the Manhattan Distance becomes larger than one step, it is more likely not to be updated with whether the coordinate is taken or not. Therefore, it is better to keep the joining peer a within a single Manhattan step of distance, as there may be another peer, c, that has allowed peer d, to join on that same coordinate, without peer a knowing this. See Figure 4.1 for a depiction of the problem.

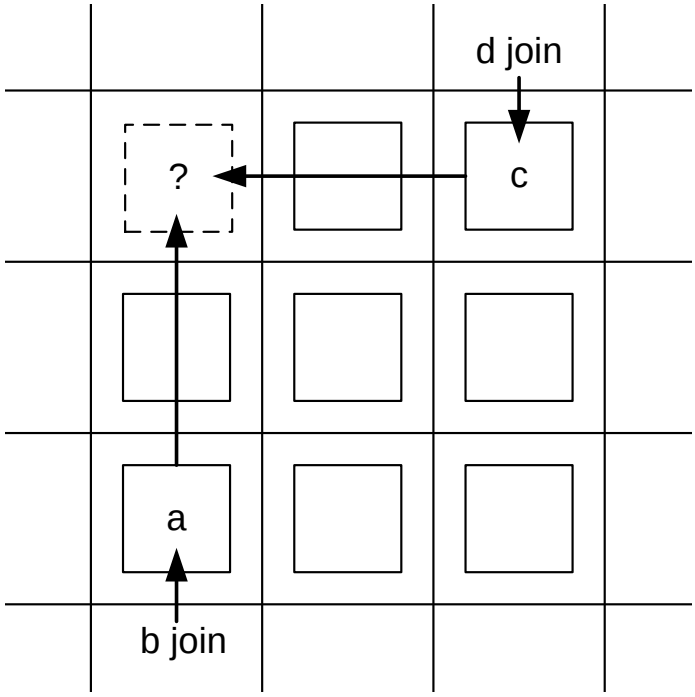


Figure 4.1: A situation that would lead to a conflict, if the algorithm allowed a joining peer to be placed as a neighbour's neighbour.

If it were allowed to join as a neighbour's neighbour, conflicts are more likely to appear, because a larger number of surrounding peers can offer the same coordinate to the joining peers b, d. Furthermore, the distance between a and c means that updates take more time to propagate to the other, resulting in the lack of synchronisation. Therefore, it is only allowed to place newly joined neighbours within a distance of a single Manhattan step.

Code Example 4.7 holds the $findNextFreeNode()$, which is responsible for locating a free coordinate a single Manhattan step away, or return a random neighbour closest to $(0, 0)$.

```
1   private Neighbour findNextFreeNode()
2   {
3     Neighbour temp = new Neighbour();
4     temp.node = null;
5     temp.timer = -100f;
6     temp.coord = new TileCoord(0, int.MaxValue);
7     List<Neighbour> tempList = new List<Neighbour>();

9     for (int i = 0; i < 4; i++)
10    {
11      if (neighbours[i].node == null && neighbours[i].coord.y
            != int.MaxValue)
```

```
12        {
13          temp = findNeighbourClosestToZero(neighbours[i],temp);
14          tempList.Add(temp);
15        }
16      }

18      if (temp.coord.Equals(new TileCoord(0, int.MaxValue)))
19        return neighbours[Random.Range(0, 4)]; // Could possibly
              cause a livelock
20      else
21        return tempList.ElementAt(Random.Range(0,tempList.Count)
              );
22    }
```

Code Example 4.7: *findNextFreeNode*() returns a free coordinate a single Manhattan step away, or a random neighbour closest to $(0, 0)$.

In Code Example 4.7, the first part (lines $3-7$) creates a temporary *Neighbour* object and resets it to values that indicate that it is empty. Lines $9-16$ check the four closest neighbour coordinates, to see if any coordinates are empty, and stores the closest empty ones in the *temp* and *tempList* variables. In lines 18 and 19, *temp* is not set (e.g. still initialized as $3-7$), if all neighbour coordinates are taken by other peers, in which case a random neighbour is returned to rejoin on. This could lead to a livelock in the unlucky situation that peers send the joining peer around in a circle, but it has not been experienced in the simulations. The livelock has solutions in Section 5.3.1 about further development, if it becomes a problem. On the other hand, if one or more free coordinates are found (more is possible because it is Manhattan Distance), a random — closest to zero — free coordinate is returned.

4.2.3 Relevant Neighbouring Peers

When a joining or rejoining peer receives a new coordinate for its tile, it also needs a list of new neighbours, relevant to the new coordinate position. The peer on which the joining or rejoining peer connects, finds a list of neighbours that are relevant to the new peer, and sends the list along with the join response. In Code Example 4.8, the method *getRelevantNeighbours*(), that creates this list of relevant neighbours, is shown.

```
1   private List<Neighbour> getRelevantNeighbours(TileCoord
        coord)
2   {
3     List<Neighbour> retval = new List<Neighbour>();

5     foreach (Neighbour n in neighbours)
6     {
7       if (n.node == null)
```

```
8           continue;

10        int dx = Mathf.Abs(n.coord.x - coord.x);
11        int dy = Mathf.Abs(n.coord.y - coord.y);

13        if (dx + dy <= 2)
14          retval.Add(n);
15      }

17      int thisDX = Mathf.Abs(this.tCoord.x - coord.x);
18      int thisDY = Mathf.Abs(this.tCoord.y - coord.y);

20      Neighbour ourself = new Neighbour();
21      ourself.coord = this.tCoord;
22      ourself.node = this;
23      ourself.timer = Statics.NODETIMEOUT;

25      if (thisDX + thisDY <= 2)
26        retval.Add(ourself);

28      return retval;
29    }
```

Code Example 4.8: *getRelevantNeighbours*() finds neighbouring peers, relevant to a specific coordinate.

Lines $3-15$ in Code Example 4.8 look through the array of neighbouring peers, and compares the neighbour coordinate with the parameter. If the neighbour is within two Manhattan Distance, it is added to the return list. Lines $17-26$ add the peer itself to the list, if it also is within the range of the two Manhattan Distance steps, before the list is returned.

Chapter 5

Evaluation

In this Chapter the test results and the simulation results are matched against the goals for the project. The Chapter aims to reason on if a Peer-2-Peer multiplayer game is a possibility for the rapidly expanding market of Smartphone games. The Chapter also points out what parts of the game need further development, to be introduced to the global market on the various application stores for Smartphones. Finally, it holds an overall conclusion of the project as a whole.

5.1 Wireless Test Results

Experimentation of sending various package sizes on two different wireless networks, was performed early in project. This was done for three reasons, first reason being that it was essential to know whether the two Android Smartphones were capable of sending information to each other. Secondly, to know if it was necessary to put more effort into the construction or the compression of the payloads that need to be sent. Finally, it is interesting to see how much delay and throughput the two different wireless connections deliver using the Smartphones.

All the tests were performed using the phones along with a purpose built application made in Unity3D. A description of how the test application was designed was shown in the Design Chapter in Section 3.3. This was to ensure that the test results were as close to the results that would be generated by the game. There were two available individual wireless networks for the tests, one of which known as "Aau-1x" and another known as "D601a". Aau-1x is the large scale optimised network at the university, this is a Cisco system with antennas and a centralised controller, which is common set-up in large-scale

public or company networks. D601a is an older small scale home type of network from Linksys (WRT54G from 2006), which was a common in households with wireless networks a few years back [MA07]. Many changes and optimisations for wireless networks means that the older D601a network is not very optimised. Given that the Aau-1x Cisco system is being updated and maintained thoroughly, the D601a network is in the case of this project, used as a worst case benchmark in relation to the Aau-1x test. All test results are available on the CD that accompanies the report.

Besides using two different wireless networks, there are two other variables to experiment with; the size of the packages and the time between sending a package. The testing software was constructed to send a number of packages with different sizes, so the tests were performed by sending 100 packages of various sizes: $\{16, 32, 64, 128, 256, 512, 1024, 2048, 4096\}$ bytes with different intervals between packages, all tests were run twice.

5.1.1 Ping Times

Looking at the results it is seen that the Aau-1x network is better optimised for the wireless network connection between the two Smartphones. This can be seen in Figure 5.1.

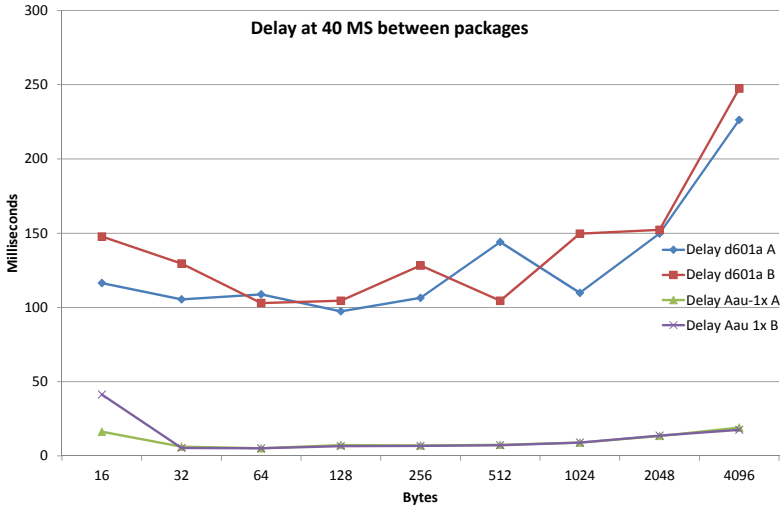


Figure 5.1: A graph showing the difference in package delay between the Aau-1x and D601a wireless networks, at 40 ms delay between packages.

Notice that in Figure 5.1 the measuring points are the average values calculated from the 100 packages sent. There are four lines because the tests were performed twice on different times on both networks. The tests were per-

formed twice, to see if the results would be the same to ensure the quality of the measurements.
An important information regarding the initial spikes in the graphs, at 16 bytes, is due to the way the testing software is created. The first time a package is received, the peer creates the UDP sender, which means that the spike represents the approximate execution time to create a UDP sender on the Smartphone.

Looking into the results gathered for Figure 5.1, the ping times of the packages lay firmly between 100 ms and 150 ms until it reaches a size of 1024 bytes on D601a. The ping times on Aau-1x lay between 5 ms and 10 ms, which is significantly better. A small rise in delay can be noticed on the Aau-1x. The reason for this increase is, that when the package payload data becomes larger than 1480 bytes, the underlying hardware layer begins to split the packages due to the Maximum Transfer Unit (MTU). This can be seen in a Wireshark package dump log. Wireshark is a tool for detailed logging of network traffic on a specific network interface [Wir11], its features make it perfect for debugging network package exchange. In the excerpt of a Wireshark package log in Table 5.1, a package split is indicated. The example holds three packages, a package with the size 1024 bytes is at the top, which is recognised as a UDP package by Wireshark and given the package number 1391. The two last packages in the excerpt (number 1393 and 1394), form a single UDP package of 2048 bytes. Notice that Wireshark sees package number 1393 as a fragment, which is reassembled in package number 1394. Package number 1393 had a size of 1480 bytes, which means that this is the maximum size, allowed by the MTU. In Table 5.1, the fields that indicate this split, are highlighted in bold.

Before commenting on whether a delay between 100 ms and 150 ms is good or bad, it is imperative to understand what it means. In games, especially fast paced games that needs approximate real-time updates on the players monitor, it is very important for the game to provide a fluid gameplay experience. Any of such fast paced games can be ruined, if the enemies do not move fluidly and skip around in the virtual world. Such a behaviour does not look realistic and it may be very difficult to e.g. shoot the enemy. This behaviour occurs when there is too much delay (too high ping time) or when packages are lost on the network. This means that the delay should be kept as low as possible for a multiplayer game. It is not possible to say what the maximum amount of delay is allowed to be, since that is a subjective point of view, it also depend on how well the game handles the delay. For instance, movement is often handled by predicting the movement based on the previous movement, for instance using interpolation, extrapolation or a technique known as “Dead Reckoning” [Aro97, VVB08]. For many multiplayer First Person Shooters, a ping time of 200 ms is denoted as a high limit. In “The effects of loss and latency on user performance in unreal tournament 2003” [BCL+04] by Beigbeder et. al.

No.	Time	Source	Destination	Prot.	Info
1391	175.620058	172.27.83.122	172.27.83.106	**UDP**	SPort 34115
Destination port: icl-twobase1 Frame 1391: 1066 bytes on wire (8528 bits), 1066 bytes captured (8528 bits) Ethernet II, Src: 7c:61:93:36:37:f4 (7c:61:93:36:37:f4), Dst: IntelCor_-05:39:03 (00:1b:77:05:39:03) Internet Protocol, Src: 172.27.83.122 (172.27.83.122), Dst: 172.27.83.106 (172.27.83.106) User Datagram Protocol, Src Port: 34115 (34115), Dst Port: icl-twobase1 (25000) **Data (1024 bytes)**					
1393	175.927076	172.27.83.122	172.27.83.106	IP	**Fragment IP**
protocol (proto=UDP 0x11, off=0, ID=0cdc) [**Reassembled in #1394**] Frame 1393: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) Ethernet II, Src: 7c:61:93:36:37:f4 (7c:61:93:36:37:f4), Dst: IntelCor_-05:39:03 (00:1b:77:05:39:03) Internet Protocol, Src: 172.27.83.122 (172.27.83.122), Dst: 172.27.83.106 (172.27.83.106) **Data (1480 bytes)**					
1394	175.928070	172.27.83.122	172.27.83.106	**UDP**	SPort 34115
Destination port: icl-twobase1 Frame 1394: 610 bytes on wire (4880 bits), 610 bytes captured (4880 bits) Ethernet II, Src: 7c:61:93:36:37:f4 (7c:61:93:36:37:f4), Dst: IntelCor_-05:39:03 (00:1b:77:05:39:03) Internet Protocol, Src: 172.27.83.122 (172.27.83.122), Dst: 172.27.83.106 (172.27.83.106) User Datagram Protocol, Src Port: 34115 (34115), Dst Port: icl-twobase1 (25000) **Data (2048 bytes)**					

Table 5.1: Table with Wireshark dumps of three packages, indicating a package split at 1480 bytes.

it is concluded that a latency of a 100 ms is noticeable while latency at 200 ms becomes annoying for the player, in Unreal Tournament. Therefore it is reasonable to aim for a upper limit of ping time on 150 ms for TileShifter, which should be possible given the relationally rather low ping on the Aau-1x network.

5.1.2 Throughput

Delay is not the only information that can be read from the test, it is also possible to determine the maximum bandwidth, also known as throughput, on the network. Both the D601a and the Aau-1x are 802.11g connections which suggest a theoretical data rate of 54 Mbit/s or $\frac{54}{8} = 6,75$ MB/s. This throughput should not be expected in practice, it is more likely half or less than half the theoretical data rate [KRA08]. Among the reasons for this decrease in data rate is, compatibility with 802.11b[1], which supports a theoretical data rate of 11 Mbit/s [WCLH05]. Furthermore, timing delays used to avoid intersymbol interference, which is a type of wireless interference that can be viewed as noise created if data collides. Distance and obstacles between device and antenna also play a role, which results in a decrease in signal strength. Finally, the wireless antennas are half-duplex, which means that the antennas can either send or receive at a given time, not both at once.

The throughput is calculated using the formula 5.1, where RW is the Receive Window (65.535 bytes), RTT is the Round-Trip Time (from device to device and back in this case) and TP is the Throughput.

$$\frac{RW}{RTT} \geq TP \tag{5.1}$$

On D601a the throughput was calculated to be approximate between $2 - 3$ Mbit/s, slowly descending toward 1 Mbit/s as the package size increases. The throughput of D601a is therefore rather low, and the graph does not pose much interest. However, the result of the Aau-1x network can be seen as a graph in Figure 5.2.

Notice, that in Figure 5.2 there is a single special case, that limits the throughput of the connection to approximately 6 Mbit/s. This is a case where another device that supports only a 802.11b is connected to the antenna, this has a bad effect on every other connecting device on that particular antenna, since it limits all connections to a 802.11b connection [WCLH05].
In the graph in Figure 5.2, every line indicates a single test with a chosen delay between the 100 packages per size.
Looking at the other test results, the throughput is above half of the promised bandwidth of 54 Mbit/s up to a size of 512 bytes, which is better than initially expected, thus making it a fairly good connection for the game. The results indicate that a size up to 512 bytes for a single package, could be a reasonable aim.

[1] 802.11a support 54 Mbit/s, 802.11b support 11 Mbit/s, 802.11g support 54 Mbit/s with average about 22 Mbit/s and the new 802.11n support $54 - 600$ Mbit/s [iee09].

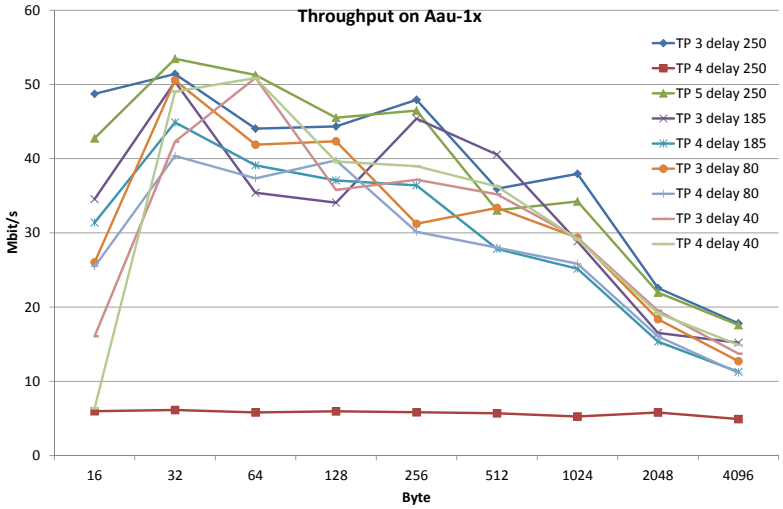


Figure 5.2: A graph showing the throughput on the Aau-1x network.

5.2 Simulation Results

In order to validate the decisions taken when designing the tilesystem and the algorithms, that form the basis for the project and TileShifter, a series of simulations have been executed. This has been done to evaluate the success of the algorithm during test cases deemed realistic, as well as how the algorithm performs under various levels of stress generated by joining and leaving peers. The many simulations performed have resulted in a multitude of simulation logs, which have been examined and evaluated, based on two different categories: Realistic test cases and Stress test cases. Both of which provide relevant results to evaluate how the algorithm, that has been developed, performs. All simulation results are available on the CD that accompanies the report.

5.2.1 Result Evaluation

Evaluating these many log results has been a challenge, as the question arises on how to judge the results of the simulations. The choice was ultimately to attempt to measure the amount of splits that each simulation produced. However, the way the algorithm is constructed, it is difficult to accurately judge when a split has occurred, as the split's tiles will attempt to rejoin to the main game if they have been cut off, with one exception: If they are able to move themselves such that they are in the coordinate $(0, 0)$, the tile will assume that it is, in fact, the correct game running, and it and any tiles connected to it will not attempt to join other games any further. Attempts to fix these

occurrences in the algorithm are discussed in Section 5.3.1. It is also difficult to know from the log if the algorithm simulation would eventually have solved splits by itself, as two tiles in $(0, 0)$ that are unknown to each other, could be connected by a long line of tiles created by leave events.
It is important to note, however, that this way of counting splits does not take into account game splits existing outside of $(0, 0)$. But due to the gravity of $(0, 0)$, they should attempt to move closer and closer to this coordinate until the split cluster eventually reaches it and becomes a real split, or runs out of tiles to move and starts to dissolve because of the lonely criteria.
For these reasons, it has been chosen to evaluate log results as follows: Look for splits, where more than one tile is currently residing in $(0, 0)$; Once one such split has been found, follow the log events until this split has disappeared and the simulation contains fewer games; Examine timestamps of this event and see if the timestamps coincide with a leave event in the log file. If so, a split has occurred, with the exception of cases where the split and the leave event are less than 15 seconds apart, due to giving the algorithm a chance to at least discover the changed game state before reaching a conclusion, based on current timeout values.
It is then pessimistically concluded that any of these splits, only removed by leave events, are splits of the game into more games, that the algorithm itself was unable to solve. It is possible, that these splits could have been solved by the algorithm if left undisturbed by further leaves, but by drawing pessimistic conclusions to these scenarios, this results in a worst case approximation.

```
⋮
7/3/2011 11:31:46 PM 26 nodes left the game. Percentage 40% of total 65
7/3/2011 11:31:59 PM There are now 2 games running
7/3/2011 11:32:26 PM Node 588 at position -2,-1 was joined by 9 nodes
7/3/2011 11:32:30 PM Node 591 at position -1,2 was joined by 9 nodes
7/3/2011 11:32:32 PM Node 612 at position 1,2 was joined by 0 nodes
7/3/2011 11:32:56 PM Node 594 at position -1,3 was joined by 1 nodes
7/3/2011 11:33:10 PM There are now 1 games running
7/3/2011 11:33:10 PM 24 nodes left the game. Percentage 41.37931% of total 58
⋮
```

Example 5.1: Log excerpt of a split.

Example 5.1 shows part of a simulation log indicating what is typically counted as a split, in a simulation evaluation. Here, a split is caused by the first leave event and two split games is the output of that leave. The split is then resolved by a second leave event and not by the algorithm itself. This counts as a split.

In Example 5.2, a different scenario can be observed. This time a split into two games is again caused by a leave event, however, the algorithm repairs it. This can be deduced because the log shows that one game is running again,

```
⋮
7/3/2011 11:25:44 PM 19 nodes left the game. Percentage 51.35135% of total 37
7/3/2011 11:26:05 PM There are now 2 games running
7/3/2011 11:26:50 PM Node 585 at position 1,-2 was joined by 1 nodes
7/3/2011 11:26:51 PM Node 585 at position 1,-2 was joined by 9 nodes
7/3/2011 11:27:01 PM There are now 1 games running
7/3/2011 11:27:10 PM Node 485 at position 1,-1 was joined by 0 nodes
⋮
```

Example 5.2: Log excerpt of the algorithm merging two games.

without the help of a leave event.

```
⋮
7/6/2011 12:29:55 AM Node 869 at position 1,-6 was joined by 1 nodes
7/6/2011 12:30:50 AM There are now 2 games running
7/6/2011 12:30:53 AM There are now 1 games running
7/6/2011 12:30:53 AM 27 nodes left the game. Percentage 30.68182% of total 88
⋮
```

Example 5.3: Log excerpt of a split falling to 15 second limit.

A log excerpt showing what falls under the 15 second criteria can be seen in Example 5.3. Two games are running, but is fixed within 15 seconds. This means that there is no way to discern if this is a split, or a join conflict salvageable by the algorithms.

Graphical explanations of some splits, can be seen in Section 5.2.4.

5.2.2 Realistic Test Cases

Realistic test cases are scenarios where the game is intended to be commonly played. In these scenarios, the algorithm should very rarely result in any splits. In these realistic cases, joins and leaves are separated by a significant timer, chosen based on the scenario. These scenarios could be playing on public transportation with friends or other passengers, playing at the university in between class, breaks or other periods with medium to long waiting times. Of course it is also entirely possible to meet up with friends just to play TileShifters.
With these considerations in mind, the simulation values have been chosen as follows: The amount of peers able to join and leave each event is relatively low, it is unlikely that a large group of people will leave unless everyone leaves. Peers joining also do this in smaller groups. The average time between these leave and join events, in other words; the intensity of the events, has been

chosen to be relatively large, since it is unlikely that peers will leave and join very often. More likely is a group of peers making a game for a length of time without leaving or joining unless by accident. The mean population of peers in a game has been chosen relative to the scenarios defined, such as 20 players on a bus or 30 players in a lecture break. The simulator will then attempt to keep the simulation mean population close to this number.
With few peers leaving and joining simultaneously, and large average intervals between each of such events, the algorithm is expected to behave splendidly and only ever result in any splits if the amount of leavers amounts to 12 or more, given that this can completely isolate peers by completely removing a neighbourhood of 12 tiles. In very unlucky situations, less than 12 peers leaving simultaneously could generate splits, this requires the tiles to build a map of specific form, which should occur rarely. Examples of such situations are shown later in Section 5.2.4.
The results of the realistic simulations can be seen in the Tables 5.2 and 5.3.

Join Leave # denotes the maximum amount of peers that can possibly join or leave every join or leave event.

Join Leave λ means the intensity of the Poisson Process used to generate the time elapsed between each event, which is used by the simulator.

μ is the population mean of the simulation, which is the amount of simultaneous peers the simulator will attempt to keep in the game at the same time.

Total Splits counts the total number of splits detected during the simulation.

Minutes is the total length of the simulation in minutes.

Splits Per Hour calculates the approximate amount of splits that would occur every hour.

Table 5.2 shows measurements from simulations that have been performed with a range of 3 to 9 maximum joining or leaving peers per event, and a relatively long 300 seconds average between each event for most simulations. The mean population aimed for is either 20 or 30. Each simulation has been run for 8+ hours. The expected results for these simulations were not to have any or very few splits occurring, as the amount of peers able to leave simultaneously were too few to completely wipe neighbour connections out. In addition the long length between each event should allow the algorithm to settle before the next leave event. Looking at the actual results, it is clear that the simulations performed even better than expected, resulting in no splits across the board. With these usage scenarios, fitting cases such as the bus, train or course room, the algorithm should perform admirably, letting all the players continue to

Join Leave #	3	3	6	6	9	9	9
Join Leave λ	300	300	300	300	300	300	1800
μ	20	30	20	30	20	30	30
Total Splits	0	0	0	0	0	0	0
Minutes	655	510	573	490	500	594	571
Splits Per Hour	0	0	0	0	0	0	0

Table 5.2: Results of realistic test cases.

Join Leave #	12	12	12	12	12	15	15	15	20
Join Leave λ	300	300	600	600	600	300	600	600	300
μ	30	60	20	30	30	30	30	30	60
Total Splits	2	0	3	2	0	6	1	2	2
Minutes	482	474	584	390	480	456	426	475	499
Splits Per Hour	0,25	0	0,31	0,31	0	0,79	0,14	0,25	0,24

Table 5.3: Results of realistic test cases.

stay connected and play together regardless of small groups of players joining and leaving.

The results shown in Table 5.3 are based on larger number of peers able to leave and join the game at the same time, from 12 up to 20 maximum simultaneous joins or leaves. The intensity of the Poisson Process ranges from 300 to 600 seconds. The mean population for the simulations has been aimed between 20 and 60. With these settings, it is expected to encounter some splits, as 12 peers leaving at the same time are able to completely isolate players, even if their neighbour lists are fully populated. However, with the lengthy average between these events, many such occurrences should be able to be sorted by the algorithm, unless the isolated group is able to move into $(0, 0)$. The actual result shows numbers in line with these expectations, as splits have occurred in a modest amount in nearly every simulation. It is actually surprising that more splits have not occurred in the simulation with 20 peers able to join or leave at once. There is a certain degree of randomness in the simulations, however, both from the Poisson Process and from the randomly generated amount of joiners and leavers, and which peers are chosen to receive joiners or leave the game.

Occurrence of splits highly depends on which peers are chosen for leaves and joins, and how the tilemap has been constructed for the particular game. Therefore, the evidence is not conclusive, but points to the algorithm and game performing well, under the intended usecases that the game is expected to be used in. The only way to know for certain, is releasing the game to the consumers.

5.2.3 Stress Testing

The stress testing simulations have been performed to see how well the algorithm performs under heavy pressure of constant joins and leaves, where the leaves are a certain percentage of the number of players in the game. These percentage based leaves could very well result in several split games, just from one leave event, as there are no bounds on the population for these tests. Peers will keep joining, which could lead to games with hundreds of players in them before a leave event occurs, shattering the grid.
These tests are expected to result in splits, probably many and often, again depending on which nodes are randomly chosen for joins and leaves, and how the tilemap has been constructed. The purpose of these tests is to examine how the algorithm behaves under severe stress, with relatively short time between join and leave events, and with increasing numbers of leaves. These scenarios are unrealistic for normal game purposes, as players will most likely not leave based on percentages, nor so often. It might, however, reveal how robust the algorithm is to churn.

In Figure 5.3, these stress simulation results have been drawn as a graph, which shows the increasing percentage of peers each leave event causes, and how many splits per hour this results in. The settings for these simulations have been a join event intensity of one every 20 seconds, while the leave event intensity is lower at one every 180 seconds. The leave percentage ranges from 20% to 60% peers leaving per event, out of the total amount of peers currently in the game.
With these settings, it is expected to encounter splits in every simulation, as the often and large leaves can very well fragment the tilemap and isolate peers in small clusters, depending on which peers are chosen to leave. However, with the often leaves, the algorithm has little time to attempt to cope with a leave event before another can occur, resulting in possible fix scenarios being evaluated as a split.

The actual results in Figure 5.3 show the average splits per hour being relatively low for the 20% leaves, but increasing up to a more severe approximately 5 splits per hour at 40% leaves, which clearly shows that more splits occur under stress, than under realistic scenarios. It should be noted, however, that split games are still playable, these games will just act as smaller games, so while undesirable, splits are not catastrophic for the game experience. Ways to improve handling of split games can be read in Section 5.3.1 about further development.

A final stress test was performed, in which 1000 peers were joined to a single game session, this conveyed no problems. The session expanded healthily and adjusted the tiles of peers, which could be placed better. Disconnecting 500 peers afterwards, resulted in 4 simultaneous games, one of which was resolved

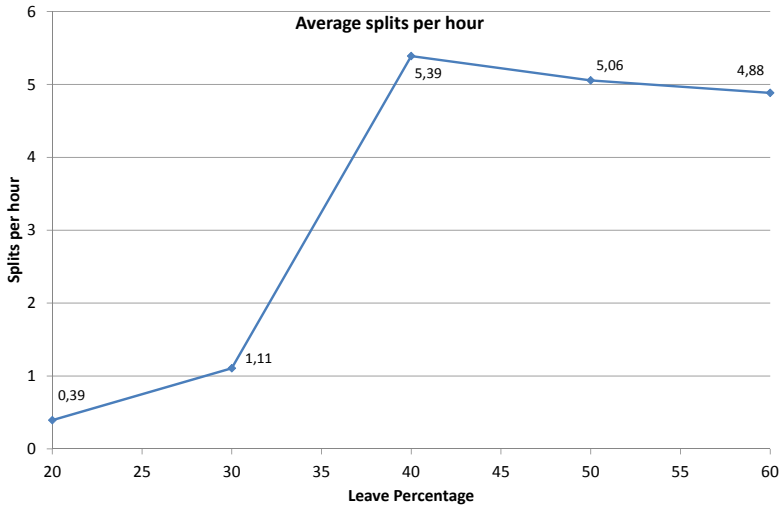


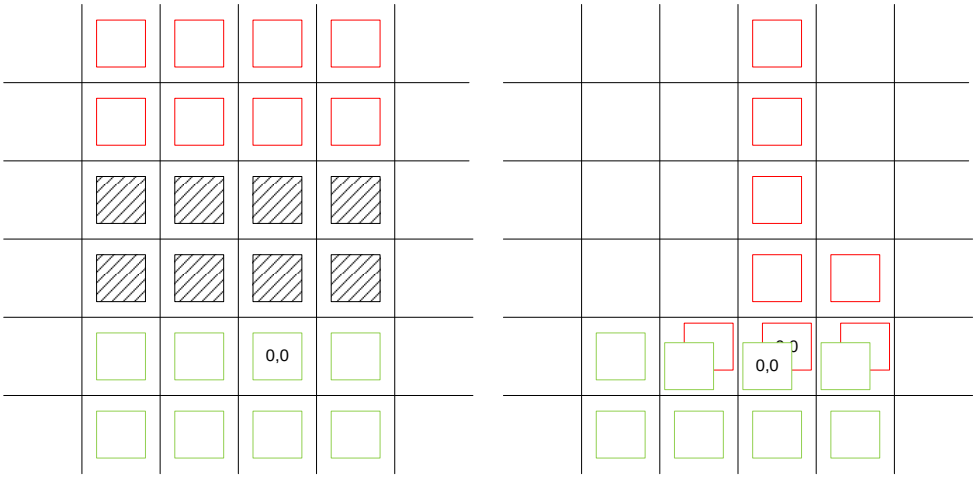
Figure 5.3: A graph of the relationship between leave percentage and average splits per hour.

by the algorithms and it ended on 3 simultaneous games. This indicates that the tile algorithms are scalable to numbers well over realistic scenarios.

5.2.4 Split Scenario

Due to the design of the algorithm, it is possible to predict some of the scenarios which would cause one or more splits to occur in a game. These are scenarios where the game is split up, such that two or more smaller clusters of peers are disconnected and have no neighbours reaching each other. In addition, these clusters should contain enough tiles to be able to shuffle and move in the grid to reach $(0, 0)$, if this happens, the splits will no longer consider themselves as a split and consider themselves an independent game session. If the clusters are unable to reach $(0, 0)$, then the closest peer will eventually consider himself lonely and try to join a closer peer, based on the global score messages received before the game was split. Without these, the cluster would still remain as a split game. An example of a split scenario can be seen in Figure 5.4.

The first Figure, 5.4a, shows a tilemap in the grid built from several peers. However, the game is hit by several leaves, and all the tiles marked with diagonal lines depart the game. This results in the red tiles and the green tiles being separated from each other. In this case, the green tiles are perfectly content being their own game and are none the wiser of the isolated red tiles. The red tiles will begin moving towards $(0, 0)$, as the tiles detect open grid positions closer and closer to $(0, 0)$.



(a) Marked tiles leave, green and red seperated.

(b) Red moves into middle, two games running.

Figure 5.4: Figures showing a split scenario.

Eventually the red tiles reach $(0, 0)$, which is shown in Figure 5.4b, and now the red tiles consider themselves to be a game of their own, none the wiser of the green tiles, that was once part of the same total game as the red ones. This scenario will cause a split, and the algorithm is unable to repair it, as both splits consider themselves an independent game.

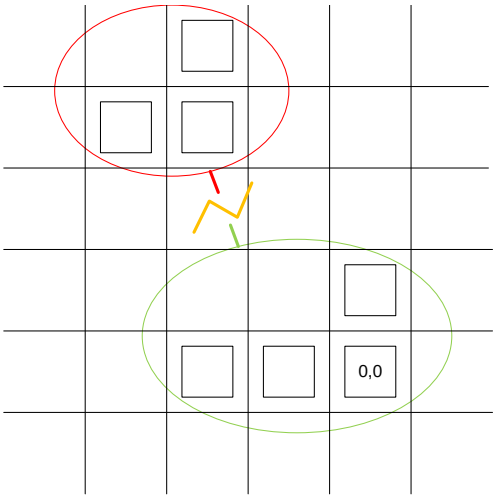


Figure 5.5: A split where the red tiles do not know any green tiles.

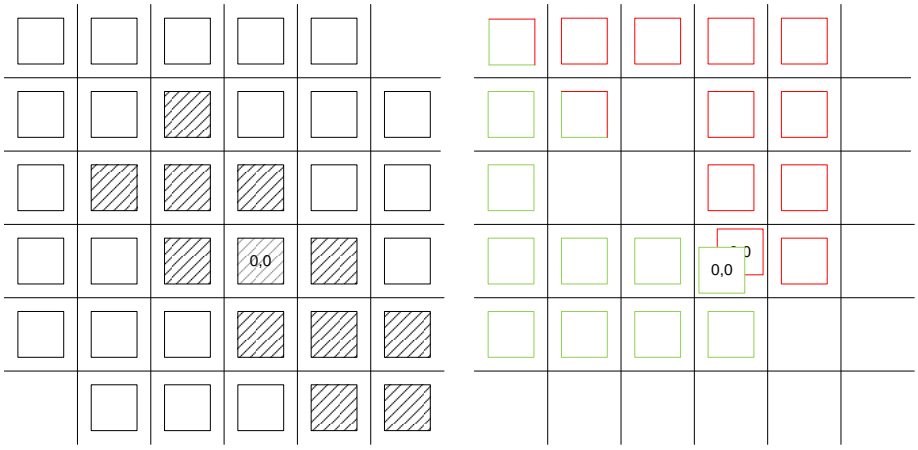
A different split scenario is the one shown in Figure 5.5, it shows a situation that has arisen after several peers have left the game, leaving the red tiles and

the green tiles isolated. However, in this situation, the red tiles are unable to move into $(0, 0)$, thus the tile closest will eventually exhaust its lonely timer. Normally, this would cause the peer responsible for the tile to look through its list of peers obtained from the global messages, but in this specific case, none of the red peers have any of the green peers in this list. This results in the red peers becoming stranded in their own game. The scenario depicted here is rather unlikely, as it requires no global messages to reach the peers responsible for the red tiles, but also requires the tanks controlled by these peers to be on red tiles. If any red tank is on a green tile, this should allow the red cluster to connect to the green cluster and eventually reassemble into one game.

Another scenario that occurs, is the situation shown in Figure 5.6, which, if given enough time, actually resolves itself, even though the simulator would read it as a split for a long time before it fixes itself. Figure 5.6a depicts a well populated game, where a large amount of peers in the center of the tilemap all leave the game, which leaves a large gap in the middle. This results in the two "ends" of the long chain of tiles wanting to move into the $(0, 0)$ grid coordinate. Due to the leaves separating them, these two parts of the game are not aware of each other and do not cause a conflict. This is depicted in Figure 5.6b. Here, the red part and the green part have both laid claim on the middle. The entire tilemap has not yet coped with the amount of leaves before, which has left a gap that has yet to be filled, allowing this split to happen. The tiles located in the top left of the grid will steadily move closer towards $(0, 0)$, and red and green tiles will come to meet each other, and discover that they belong to the same game. This will cause any conflicts caused by the green and red tiles occupying the same coordinates, to eventually resolve themselves, including the two occupants of $(0, 0)$, turning the split into a single game once again.

5.3 Further Development

If TileShifter was to be distributed and sold in an AppStore, some further development is needed to perfect the product. Furthermore, during the development, some possible improvements for the network overlay algorithm have been discovered. Therefore, this Further Development Section accentuates potential improvements for both the overlay algorithm and TileShifter as a game. The following Sections divide the further development into categories of whether it is on the tile algorithms in Section 5.3.1, for the game in Section 5.3.2 or the networking between peers in general in Section 5.3.3.



(a) A populated game suffering leaves.

(b) Red and green claim the middle but are really the same game.

Figure 5.6: Figures showing a split scenario that eventually solves itself.

5.3.1 Overlay Algorithm

The sole purpose of the tile algorithms, is to keep tiles connected as best as possible, to form an exciting map in the game world. The purpose is also to avoid splits in the communication, which results in the creation of smaller separate games as much as possible, when peers leave or disconnect. Although, as the simulation results in Section 5.2 show, the algorithm avoids many splits and repairs potential problems very well, but splits do still happen. The biggest obstacle is that no entities have knowledge or overview of the entire network in a Peer-2-Peer network, thus, this problem is impossible to avoid entirely. With more tweaking and a better discovery of splits, however, the algorithm could be improved to resolve more of these situations.

Join Livelock

The livelock noted in the *Join*() algorithm in Section 3.2 is unfortunate, but is unlikely to happen, as no simulation ever encountered it. It can become a livelock, because the joining peer can be sent in circles. If it becomes a noticeable problem, then it can be solved by not returning any random neighbour, or returning a random neighbour until a join timeout expires. The algorithm will instead return a random neighbour picked from neighbours further away from $(0, 0)$, than the tile the join was attempted upon.

Rejoining On Splits

A proposed improvement is to utilize more information, available about other peers. For instance, if a tilemaster is moving on another tile, it can rejoin the other tilemaster if it finds itself isolated. This also works the other way around, if the tilemaster drives on its own tile, it may rejoin on a peer that drives on his own tile. This leaves three possible problematic situations:

Peer is alone on own tile A peer can be a tilemaster on its current tile, and no other peer is on the tile. In this case the solution does not help.

Other peers to rejoin on may be isolated themselves If there are peers to rejoin on, there is no guarantee that that particular peer is not isolated as well. But atleast it is likely, that the peer will be able to continue an unresolved battle with known peers.

Other peers may not be tilemasters themselves To heighten the action experience, the number of tiles in the game could be limited i.e. to half of the joining peers. This can be used to ensure that replication does not put too big a burden on a single peer, and it reduces the size of the world, so that tanks meet more often. The trade-off is that there are less peers to rejoin on, in case of isolation.

The number of splits may be reduced, but the problem will still occur. Finally, the isolated tilemaster can try to rejoin on the peer it originally joined on, if it still exists. In this situation, the two peers will still be connected to the same game, which means that the player is less likely to notice the split, based on the assumption, that they joined each other because they want to play against each other.

Discovering Splits

Improving the discovery of splits as separately running games, can be done using the coordinate system and neighbours. Think of the peers in the coordinate system as routers with a coordinate identification, this way a peer would know if a message should be propagated to the north, east, south or west to reach a specific coordinate. That knowledge will aid in the area of pathfinding and indication of directions towards enemies, and in connection with detection of splits. It can be used to detect if it has connection to $(0, 0)$ or to the tile of a peer that drives on its own tile. It can also be used the other way around, where the peer drives on another tile, it can check if there is a connection between the two.
In practice, this could be implemented such that a tilemaster sends a message

towards the coordinate it wants to check, and wait a predefined timeout for response. The message should hold the destination coordinate, and the IP of the sending peer, such that an answer can be sent directly back to the sender upon success. The propagation could work in much the same way as a depth first search pathfinding algorithm. If the timeout expires without response, the peer must expect that it cannot reach the destination, and a corresponding action must be taken. The timeout value should depend on the Manhattan Distance between the two coordinates.

5.3.2 TileShifter

The game itself must be polished and the missing features implemented. Further development on the TileShifter, is mostly cosmetic changes and additions, but also ideas to expand on the gameplay.

Cosmetic Changes

In connection with this project, cosmetic changes are not important, but if TileShifter were to be distributed and sold, cosmetic problems and issues can easily be the difference between success and failure, for a small action game such as TileShifter. Consumers are hard critics and a few bad reviews can have a huge impact on the interest gained by other consumers. If, however, the game looks good, feels good and presents itself well, more consumers are going to give it a chance. The game therefore, must be polished in the following areas:

- Tile textures should be finished
- More tiles should be added
- Better user interface
- A menu should be added
- More feedback, such as explosion sounds, particle effects and device vibrations

Play testing can signify whether the game can be a success or not, before distributing it on the market.

Missing Functionality

Also in terms of functionality, a few implementations are missing before the game is finished. The missing functionalities in this Section are all viewed as necessary to deem TileShifter a complete game. The functionalities are briefly described below:

Player Names Being a competitive scalable game, usernames should be supported, such that the players can see which opponent they are firing at and competing against.

Scoring Global messages containing scores and player names should be propagated on the network, such that the players can view their performance relative to others.

Guidance Arrows Arrows that point in the direction of a number of the nearest enemies should be implemented. These could be generated by a pathfinding algorithm and point in the direction of another tiles coordinate, if there are no other players on the current tile.

Tile Replication The functionality of tile replication is needed to save players moving on a tile with no tilemaster. Remember, that the tilemaster works as a server for the single tile, so if this tilemaster leaves the network with other players on the tile, another peer must be regarded as the master. Therefore, a player moving on a tile, has to replicate the master of that tile.

Cross Platform To reach a larger consumer market, TileShifter must be distributed on more platforms. Using Unity3D, it is very easy to distribute on Apple iOS and Sony PSP as well. This will only need minor changes in the input handling, since Unity3D takes care of the translation to native code.

Expanding Game Concepts

During the development of TileShifter, a number of additional alternate game modes have been envisioned, that can improve the game experience for the players. This may be small additions and entirely other game modes or ways to play the game. This can be controlled when the game session is created initially. Which game modes are activated and the information of the game mode is given at join, any time a player joins, all peers will play the same game mode chosen, provided that it cannot change for the entire game session. In the description below, a few such additions and alternate game modes are described:

Power-ups Players may be able to pick up power-ups. This could be better weapons, more ammo or repair-kits.

Team Game Players are divided into teams and play against other teams. Scores are the same for the entire team, and friendly fire subtracts from the score.

Quizmaster Tanks Periodically ask a question, requiring the players to move to specific tiles to answer. This is a good way of distributing load on different tiles, thus lowering bandwidth use on a single tilemaster.

Custom Tile Placement When joining, the player can choose where to place his own tiles in the world, maybe buy specific tiles for experience points earned during gameplay.

Custom Tiles Support creation of custom tiles, which could be built from either predefined pieces or player created content, provided it lives up to criteria such as having transitions in the correct locations. A player would be able to place a custom built tile, when joining a game session.

5.3.3 General Networking

Finally there are a few networking features and optimisations that need implementation. The improvements range from important missing necessary implementations in the current state, to ideas on how to improve the networking for TileShifter.

Global Messages

Initially, the most important part is to finally implement and fine tune the global messaging. These are messages containing scores and player names and work as keepalive messages. They should also be used to find another tilemaster to connect to, if a player becomes isolated. This means that this is the most important feature missing in the implementation, since it is one of the cornerstone functionalities of connection recovery in the overlay algorithm.

Package Optimisation

Part of optimising the networking, is the ability to handle missing packages or packages caught in high latency. When an enemy moves on the screen, it must be as fluid as possible, and this can demand a lot of bandwidth when movement updates are sent frequently. Furthermore, the latency might be high on the network or the movement packages could be lost. To be able to handle

this, the game must be able to perform predictions on movements of enemies, which could be done by extrapolation. When a package containing a movement update arrives within a short time span, often set to the average latency ping time, the client can use interpolation, and move the enemy according to the position in the package. In the situations, however, where an update is lost, or arrive late, the client still needs to know where to move the enemy player's tank on the screen. In such cases, extrapolation or dead reckoning is used. Extrapolation uses the last updates to calculate and predict the direction the enemy is moving, while dead reckoning predicts where the enemy is moving using direction and velocity. This concept could be a great improvement for TileShifter.

As part of the network optimisations, data encoding or data compression could be a potential improvement, as previous research has indicated on for example the HTTP [MDFK97]. Data encoding covers the concept of only transmitting changes or differences between new and old updates. Compression of the data is another way of saving bandwidth, but this is only relevant, when the size of packages exceeds 512 byte (as discovered in Section 5.1 with test results).

In TileShifter, messages received by a peer is buffered in a queue. This has proven to be helpful in connection with the threads that run the game and the thread that receives data on the network. However, as mentioned in "A Multiplayer Real-Time Game Protocol Architecture for Reducing Network Latency" by Ahn et. al. [wACBF09] implementation of a protocol that discards outdated packages could be an improvement. The idea is, that any server (or tilemaster in the case of TileShifter), receives a lot of updates, and as they end up in a buffer, it may receive updates that render older updates in the buffer obsolete. In this case, it is best to ignore and disregard this update to avoid using computational power or bandwidth on distributing the obsolete data to the other peers on the tile.

Extra Research

Dealing with a mobile platform, not only resources are limited, but the battery power also sets a limit for the gaming experience. Games generally use a lot of computational power, this means that the games also use a lot of battery power or energy. When using the networking interfaces on the Smartphone as well, it consumes the energy even faster. Since energy is a limited source in a Smartphone, it is a good idea to think of the energy consumption. Anand et. al. [ASM+11] have introduced a protocol that is energy saving and developed for network gaming on mobile devices. This is a very relevant field to improve on in relation to networking in games for Smartphones and it is no different for TileShifter.

Finally, the use of a MANET could be a way to enhance the connectivity between peers. A wireless router might not be available and some wireless routers can be set to block Peer-2-Peer traffic. In such cases, a MANET could be a solution. The MANET was shortly described in Section 2.3.1. The connections of a MANET could be better integrated with the tile system. For instance the physical location of peers could control tile and neighbour locations in the game world.

5.4 Conclusion

The primary goal of this project, as formulated in the Problem Statement in Section 1.1, was to examine the possibilities of developing a Peer-2-Peer multiplayer game, for a Smartphone environment. In a Peer-2-Peer network, many problems arise, such as connectivity stability, synchronisation and randomly disconnecting peers, but if these obstacles can be traversed, the Peer-2-Peer multiplayer game will be serverless and highly scalable. Game development is often a task with great freedom of choice, so it is proposed to focus more on game design-driven development. The idea is to use game design to solve functionality– and network– related problems, to highest possible extend.

5.4.1 TileShifter

The result of the project is TileShifter, a Peer-2-Peer multiplayer action game. The gameplay of TileShifter is simple, players control tanks, driving around and shooting each other in a tile-based and randomly constructed world. Tiles are used to solve most of the Peer-2-Peer network related problems, using a series of custom algorithms, described in Sections 3.1.1 and 3.2, created specifically for tile-based games. Although, the tile algorithms are created for TileShifter, it is meant as a generic set of algorithms, usable for other game types, that can be based on tiles.

For TileShifter to be a success, as an implemented Peer-2-Peer multiplayer game, a set of requirements were formalised in the Problem Statement in Section 1.1. The following description re-lists the demands for TileShifter, and points out the features that ensure the demands are fulfilled.

Peer-2-Peer TileShifter uses Peer-2-Peer and does not require a dedicated server or tracker of any kind. The peers connect directly to each other using UDP connections.

Smartphone Platform TileShifter is developed for Android, and is tested on the HTC Desire. Using Unity3D, it is possible to compile the game

for Apple iOS as well.

Scalability Scalability is achieved in TileShifter, through the use of the grid-based tile system, which expands with every new peer. The tilemaster concept provides a semi-server for peers on a tile, which limits the communication to interested peers only.

Robustness The tile algorithms takes care of cleaning and maintaining the connections between peer owned tiles, such that players can join and leave without issues. Therefore, the tile algorithms ensure the robustness of TileShifter.

Drop in/out Gameplay Action is not far away, due to clustering of the tiles in the virtual world, since it makes sure that distance between tiles is minimised to a certain degree.

Game Driven Design The tile-based algorithms have proven to be very useful in connecting gameplay features with functionality and connectivity between peers. Moving tiles both repair connections and stimulates explorative gameplay.

The demands for TileShifter have all been acknowledged in the implementation, and as such, it is successful in relation to this project. As noted in the description above, most of the success lies in the tile algorithms, so next step is to conclude on how successful the algorithms have proven to be.

5.4.2 Tile Algorithms

As for TileShifter, certain demands of the tile algorithms were formalised in Section 3.2. The tile algorithms are an important factor in this project, since they propose a new way to organise a Peer-2-Peer game, based on tiles. Due to their importance, it is crucial that the demands are met, therefore they are evaluated in the following description, based on the simulation results in Section 5.2.

Robustness When a peer joins, other peers are not affected at all, since the joining peer is placed in a free coordinate and adjusted to a better position, if possible. When peers leave, the tiles rearrange in the grid and other peers can still play the game. Even when the tiles are separated and splits occur, TileShifter is still playable for the isolated peers. Furthermore, they are most likely still connected to the peers they have had encounters with, due to the way an isolated group of peers rearrange their tiles, based on what other tiles they drive on and have received global score messages from.

Reachability Rearrangements of tiles in the grid and clustering of tiles, ensure that tiles are connected and that the distance between tiles is as small as possible. With the exception of the unlikely problem with long lines of tiles forming outwards from $(0, 0)$, instead of clustering, which was mentioned in Section 3.1.1.

Scalability The scalability of the tile algorithms has been tested with several realistic simulations, based on real-world usecases, such as a class break or during public transportation. The test results were good, but splits of games could still occur. Percentage based leaves illuminated this, but join tests showed that the algorithms could easily let thousands of tiles join at least as described in Section 5.2.3.

Decentralised Although a tilemaster is used as a localised server, there are no need for a dedicated machine to act as server or tracker. Therefore, the tile algorithms work independently, making TileShifter a decentralised Peer-2-Peer game.

Since large scale tests of TileShifter have been unreachable, due to lack of test devices, we have had to rely solely on the results of the simulations. The simulations in Section 5.2 indicated that the tile algorithms perform well under realistic situations, while stress testing indicated, that there is a large increase in splits per hour from 30% to 40% leaves. Intelligent selection of which peers to attempt to rejoin on, in the event of a split, would mitigate ill effects a split could have on the player. This is because the player's tile would reposition according to recently encountered tanks.

Being satisfied with the performance of the tile algorithms and TileShifter itself, the final performance question of the wireless network still needs answering.

5.4.3 Wireless Testing

In Section 5.1 about Wireless Test Results, two areas of wireless network connections were tested for two different networks. The tests were performed by sending packages of increasing sizes from one Smartphone to another, and back again.

The first test results determined the network latency — or ping time — on the two wireless networks. The latency indicates that if the wireless network is optimised for connections with Smartphones, then the ping time has little influence on the gaming experience, with a ping time of approximately 10 ms. On an older unoptimised network, latency could pose a problem with a ping time of approximately 150 ms.

The second test results calculated the throughput of the two networks, to indicate the large difference. The optimised network allowed a practical throughput of approximately 30 Mbit/s at a package size of 512 bytes. The older unoptimised network could perform at approximately 3 Mbit/s at a package size of 512 bytes, which indicates a significant difference.

Looking at the graphs of the results in Section 5.1, it can be seen that, up to the package size of 512 bytes, the optimised network could deliver more than half of the promised theoretical throughput of 54 Mbit/s. Therefore it is argued that to get best performance, the packages should aim for a size of 512 bytes. It is best to piggyback as much data in one package as possible, to save overhead on headers added to packages. It was furthermore noticed, that the underlying network layer, automatically split the packages at a size of 1480 bytes, so this is regarded as a hard limit for the size of packages. If packages becomes larger than 1480 bytes, the number of packages on the network are increased.

5.4.4 Final Words

Development of a game, is a time-consuming and complex task, and as such, many features have been omitted due to time constraints or because it is outside the scope of this project. Many of these possible improvements on the TileShifter game, were listed in Section 5.3.

Despite the simplicity of the gameplay, TileShifter is a fun action game, in the eyes of the authors. It allows a much larger group of competitors than other games available for the Smartphone, and there are many ideas for expanding the gameplay. It would be enjoyable to see the game finally developed and available for several platforms in app stores.

The initial problem of whether a multiplayer Peer-2-Peer game can be developed for a Smartphone, is answered with a resounding “yes”, demonstrated through the creation of TileShifter.

Bibliography

[App11] Apple. Apple App Store, 2011. http://www.apple.com/iphone/apps-for-iphone/.

[Aro97] Jesse Aronson. Dead Reckoning: Latency Hiding for Networked Games, 1997. http://www.gamasutra.com/view/feature/3230/dead_reckoning_latency_hiding_for_.php.

[ASM+11] B. Anand, J. Sebastian, Soh Yu Ming, A.L. Ananda, Mun Choon Chan, and R.K. Balan. Pgtp: Power aware game transport protocol for multi-player mobile games. In *Communications and Signal Processing (ICCSP), 2011 International Conference on*, pages 399–404, feb. 2011.

[Bar11] Margherita Barile. Taxicab Metric, 2011. http://mathworld.wolfram.com/TaxicabMetric.html.

[BCL+04] Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. The effects of loss and latency on user performance in unreal tournament 2003. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, NetGames '04, pages 144–151, New York, NY, USA, 2004. ACM.

[BDL+08] Ashwin Bharambe, John R. Douceur, Jacob R. Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. Donnybrook: enabling large-scale, high-speed, peer-to-peer games. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 389–400, New York, NY, USA, 2008. ACM.

[Beu05] Jan Beutel. *Design and Deployment of Wireless Networked Embedded Systems*. PhD thesis, ETH Zurich, Zurich, Switzerland, 2005.

[Blu11a] Bluetooth.com. The Official Bluetooth Technology Website, 2011. http://www.bluetooth.com.

[Blu11b] Bluetooth.org. The Official Bluetooth Special Interest Group Website, 2011. http://www.bluetooth.org.

[CDD11] T. Clausen, C. Dearlove, and J. Dean. Mobile Ad Hoc Network (MANET) Neighborhood Discovery Protocol (NHDP). RFC 6130 (Proposed Standard), April 2011.

[CDDA09] T. Clausen, C. Dearlove, J. Dean, and C. Adjih. Generalized Mobile Ad Hoc Network (MANET) Packet/Message Format. RFC 5444 (Proposed Standard), February 2009.

[CDK05] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley Professional, fourth edition, 2005.

[CE10] Young Choi and EETimes. Analysis gives first look inside Apple's A4 processor, 2010. http://www.eetimes.com/design/signal-processing-dsp/4089026/Analysis-gives-first-look-inside-Apple-s-A4-processor.

[Dan11] Hi3G Danmark. 3G Data Rate of up to 32 Mbit/s (Commercial, Danish), 2011. http://www.3.dk/Privat/Mobilt-bredband/Hastighed-og-dakning/Hastighed2/?nm_extag=datadelingpuf_32mbit.

[Goo11] Google. Android Market, 2011. https://market.android.com/.

[Hen09] Christopher Henden. Simulating the Poisson process, 2009. http://www.oddnumber.co.uk/2009/05/30/simulating-the-poisson-process/.

[HTC11] HTC.com. Official HTC Desire Specifications, 2011. http://www.htc.com/europe/specification.aspx?p_id=312.

[iee07] Ieee standard for information technology-telecommunications and information exchange between systems-local and metropolitan area networks-specific requirements - part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE Std 802.11-2007 (Revision of IEEE Std 802.11-1999)*, pages C1–1184, 12 2007.

[iee09] Ieee standard for information technology–telecommunications and information exchange between systems–local and metropolitan area networks–specific requirements part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications amendment 5: Enhancements for higher throughput. *IEEE Std 802.11n-2009 (Amendment to IEEE Std*

802.11-2007 as amended by IEEE Std 802.11k-2008, IEEE Std 802.11r-2008, IEEE Std 802.11y-2008, and IEEE Std 802.11w-2009), pages c1–502, 29 2009.

[IML+03] H. Inamura, G. Montenegro, R. Ludwig, A. Gurtov, and F. Khafizov. TCP over Second (2.5G) and Third (3G) Generation Wireless Networks. RFC 3481 (Best Current Practice), February 2003.

[JE03] JWatte and Enchantedage.com. How to use an introducer to do NAT punch-through for peer-to-peer communication, 2003. `http://www.enchantedage.com/node/8`.

[KLXH04] B. Knutsson, Honghui Lu, Wei Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 1, pages 4 vol. (xxxv+2866), march 2004.

[KRA08] J.F. Kurose, K.W. Ross, and B. Anand. *Computer networking: a top-down approach*. Pearson/Addison Wesley, 2008.

[Lei11] Alexander Leigh. Angry Birds Sees 100 Million Downloads, 2011. `http://www.gamasutra.com/view/news/33509/Angry_Birds_Sees_100_Million_Downloads.php/`.

[LS10] Steve Litchfield and All About Symbian.com. Defining the Smartphone, July 2010. `http://www.allaboutsymbian.com/features/item/Defining_the_Smartphone.php`.

[MA07] Bradley Mitchell and About.com. Top 802.11g Wireless Broadband Routers for Home, 2007. `http://compnetworking.about.com/od/wirelessrouters80211g/tp/80211ghome.htm`.

[MDFK97] J.C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for http. *ACM SIGCOMM Computer Communication Review*, 27(4):181–194, 1997.

[MS99] Heidi Monson and SysOpt.com. Bluetooth Technology and Implications, December 1999. `http://www.sysopt.com/features/network/article.php/12029_3532506_1`.

[NZ09] Andrew Nusca and ZDnet.com. Smartphone vs. feature phone arms race heats up; which did you buy?, August 2009. `http://tinyurl.com/featurephone`.

[PCM11] PCMAG.com. Definition of: Smartphone, 2011. http://www.pcmag.com/encyclopedia_term/0,2542,t=Smartphone&i=51537,00.asp.

[PM11] Simon Pope and Trudy Muller. Apple's App Store Downloads Top 15 Billion, 2011. http://www.apple.com/pr/library/2011/07/07Apples-App-Store-Downloads-Top-15-Billion.html.

[Pos80] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[Pos81] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168, 6093.

[PS11] Christy Pettey and Holly Stevens. Gartner Says Android to Command Nearly Half of Worldwide Smartphone Operating System Market by Year-End 2012, 2011. http://www.gartner.com/it/page.jsp?id=1622614.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Rachid Guerraoui, editor, *Middleware 2001*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45518-3_18.

[Tel11] Telia.dk. EDGE, 3G and 4G Data Rates (Commercial, Danish), 2011. http://telia.dk/mobiltbredbaand/omhastighed/.

[Uni11] Unity3D. The Official Unity3D Technologies Website, 2011. http://unity3d.com.

[VVB08] James Van Verth and Lars M. Bishop. *Essential Mathematics for Games And Interactive Applications*. Morgan Kaufmann, second edition, 2008.

[wACBF09] Yong woon Ahn, A.M.K. Cheng, Jinsuk Baek, and P.S. Fisher. A multiplayer real-time game protocol architecture for reducing network latency. *Consumer Electronics, IEEE Transactions on*, 55(4):1883–1889, november 2009.

[WCLH05] S.-C. Wang, Y.-M. Chen, Tsern-Huei Lee, and A. Helmy. Performance evaluations for hybrid ieee 802.11b and 802.11g wireless networks. In *Performance, Computing, and Communications Conference, 2005. IPCCC 2005. 24th IEEE International*, pages 111 – 118, april 2005.

[Wei11] Eric W. Weisstein. Poisson Process, 2011. http://mathworld.wolfram.com/PoissonProcess.html.

[Wir11] Wireshark.org. The Official Wireshark Website, 2011. http://www.wireshark.org.