Abstract

Video games are very popular today and the electronic entertainment industry is in many areas equal to the more mature entertainment industries like film and music in terms of usage and revenues.

The MOOSES (Multiplayer On One Screen Entertainment System) allows players to play on one big screen together with the use of their mobile phone as a game controller. The framework was designed for easy development of multiplayer games in Java and C++.

Flash is a very popular technology on the Internet, with a great many people using and developing all kinds of games, video players and other software applications. Flash applications are easy to develop due to a modern programming language and a vast amount resources available online.

In this thesis the possibility of creating Flash games for MOOSES will be explored. A set of components will be developed which will allow Flash games to use the MOOSES framework. These components will be designed to aid the game creation as much as possible and still be generic enough to allow as many game types as possible. Three different games will be developed using the components and each game will focus on a different multiplayer mode. In the end, the components, the three games, and the Flash platform will be evaluated.

Preface

This master thesis was written by Magnus Førland Ekse in the period from January to mid June 2010 at the Department of Computer Science, Norwegian University of Science and Technology, under the supervision of associate professor Alf Inge Wang.

Acknowledgements

I would like to thank Alf Inge Wang for his help and guidance throughout the project.

I would also thank Morten Versvik for technical assistance with MOOSES.

Trondheim, June 11th, 2010

.....

Magnus Førland Ekse

CONTENTS

LIST OF FIGURES

Part I

Introduction

CHAPTER

1

INTRODUCTION

The video gaming industry has grown from single person projects distributed on floppy disks to projects developed by hundreds of persons with multibillion dollar companies as distributors. Not only has the size of the projects grown, but also the games themselves. Graphics have improved from very pixelated representations of real world objects to high resolution and photo-realistic depictions. The gameplay has evolved from simple button mashing to emotional pieces of entertainment. The amount of work needed to publish a state of the art games is now many times more than before. As a response to this, some companies has developed frameworks and technologies which allow one man projects to enjoy new technologies, rapid development and easy distribution. One example of such technology is Flash.

Originally developed as a animation tool, Flash has been improved with extensive scripting support and is today a full-fledged multimedia platform. Flash has become immensely popular technology on the internet, as almost every computer with an internet connection has Flash installed. Popular uses include video playback, animations, advertisements and games. When developing in Flash, the time from code to screen is very low, allowing for rapid prototyping and testing ideas. While larger projects are certainly possible, most Flash games are relatively simple and intended for casual gamers.

MOOSES is short for Multiplayer On One Screen Entertainment System, and it is a framework for games and interactive applications. MOOSES has been developed at NTNU as previous master's theses. The framework allows people to use their

mobile phones (although recent development allows laptops and other devices) as controllers to play games and use interactive applications together on a large, high-definition screen found in cinemas, lecture halls and convention centers.

1.1 Motivation

The author has been interested in video games ever since playing the first Sonic game on the Sega Mega Drive, a 16-bit console system for TV screens. This passion has continued to grow over the years and was one of the reasons he started learning about programming. Recently, games created by independent game developers have started to flourish, which have given even more inspiration.

Previous games for the MOOSES framework have been developed in Java, C++ and C#, but so far an implementation in Flash has not been thoroughly tried and tested. Flash is probably the most popular technology (excluding technologies natively implemented by most browsers) on the web right now. To open up MOOSES for the thousands of talented Flash programmers and artists, this thesis will look at what is required to create a Flash game for MOOSES, as well as create the artifacts needed to ease such development and test those artifacts by developing several games.

Hopefully the product of this project will serve as a help for people who want to use Flash for MOOSES.

1.2 Project Context

This report along with the games created is the deliverable of a master's thesis written at Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU)(IDI). The workload is one semester.

This project is part of a joint research program between Tellu and IDI. At IDI this is a part of the game technology research program, which Alf Inge Wang is in charge of.

Tellu is built on Ericssons former department of research in Norway. Today they further develops the technology used in MOOSES, and other issues related to mobile phone technology. Tellu is also responsible for the commercialization of MOOSES.

1.3 Stakeholders

This project has four main stakeholders, Tellu, Alf Inge Wang, Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU)(IDI) and the author of this report.

1.4 Project Goals

The goals of this project are to explore the use of Flash technology with MOOSES and look at different multiplayer modes. These goals will be achieved by implementing a set of artifacts that will facilitate the development of MOOSES games and using those artifacts to create three games with distinct multiplayer modes. The artifacts and games will finally be evaluated.

CHAPTER

2

REPORT OUTLINE

This report has been split into seven parts for easier reading. This chapter will give a brief description of each part to let the reader navigate the report more easily.

Part I Introduction This part contains an introduction to the project along with motivation, stakeholders and project goals.

Part II Research Questions, Environments and Tools This part defines the research questions driving the project, the method of research, and the test environment and development tools.

Part III Prestudy This part elaborates on some of the research material for this project, such as video games, the MOOSES framework, Adobe Flash and state of the art (previous research at NTNU and existing big screen solutions).

Part IV Flash for MOOSES Framework This part describes the modules created to aid and simplify further Flash game development for MOOSES.

Part V Game Development This part describes the concept and some of the implementation of the three games.

Part V Evaluation and conclusion This part evaluates the game, the Flash platform and the Flash for MOOSES Framework. It also concludes the report and brings up some ideas and suggestions for further work.

Part VI Appendices This part contains source code and other texts found to be too technical or unsuitable for any of the other parts of the report, but still relevant for the work done and for further work.

Part II

Research Questions, Environment and Tools

CHAPTER

3

RESEARCH

This chapter will describe what questions that should be answered to get an overview of using Flash to create MOOSES games, evaluate the player experience with different multiplayer modes, and the method to obtain the answers to these questions.

3.1 Research focus

We will explore how to create games for MOOSES using Flash, and to create reusable components for the development of Flash games for the MOOSES framework. We will also take a look at how the players experience different multiplayer modes.

3.2 Research questions

This report will answer the many questions that have arisen when attempting to use Flash for MOOSES.

RQ 1 What must be done in order to create a game in Flash for the MOOSES framework?

Research will be done in order to find out what must be created or prepared in order to successfully develop a game in Flash for the MOOSES framework.

A) How should the framework and the Flash game communicate?
B) What modifications must be done with the MOOSES framework to allow communication with a Flash game?

C) What is needed of both experience and software to develop a MOOSES game in Flash?

RQ 2 What challenges does MOOSES development in Flash pose?

Challenges and constraints by using Flash will be researched, both in game development in general and in connection with the MOOSES framework.

A) How does a Flash game perform in a high definition resolution?
B) What constraints and differences does an implementation in Flash have compared to previous implementations of MOOSES games?

RQ 3 How should the modules created in this project be organized for rapid development of new Flash games for the MOOSES framework?

Research will be done in order to find out how generalise the process of creating a game for the MOOSES framework in Flash.

A) What parts of the prototype game should be reusable components for new games?
B) What directions and recommendations for new projects should be formalized?

RQ 4 How is the user experience when playing MOOSES games?

Different games are planned to be created in order to explore different aspects of multiplayer modes; all versus all, team versus team and team versus team with shared control of the playable character.

A) Is shared control over a character fun and enjoyable?

B) How do players experience the different multiplayer modes?
C) Does cooperation work in a MOOSES game?

3.3 Research Methods

While there are several methods of doing research, Basili [HDRS93] names some approaches suitable for software engineering.

In the *scientific method*, the software process is investigated by using analysis, observation and evaluation. The process is observed and evaluated, and the results of the evaluation are used to propose improvements to the system. Improvements can be in the form of better tools and methods, or improvements in the process itself. The process of observe, evaluate and improve can be repeated until the result is satisfactory.

A variation of this method is the *engineering method*. A solution or system is improved through iterations. The solution is observed and a better solution which may solve found problems is proposed. The solution is built and then carefully measured and analyzed. This step by step approach should then be repeated until the solution is good enough.

On the other hand there is the *empirical method*, which does not necessarily depend on an existing solution. A new model of a solution is proposed, then data is collected, analyzed and evaluated, and in the end the model is validated. The goal is to study the effects of the process and/or the product. As with the engineering method, correct measurement and analysis is important, and the data-collection must be based on reason.

Finally in the *mathematical method* a theory is proposed (a set of axioms) and further developed. The results of the theory is derived and compared to empirical observations to verify the credibility of the theory.

For this project the engineering method is the most useful when it comes to creating the reusable modules as well as the prototype. A solution will be created, evaluated and improved upon until the solution is working satisfactory. Literature study (of papers, articles, books and relevant source code) and lessons learned (from previous projects and this one) will provide the method of gathering the knowledge necessary to give an answer to each of the research questions.

In order to gather and evaluate the player experience when using the created games a simple survey within the test subjects will be conducted. A questionnaire will be handed out after each playtest. There are many reasons for the use of a questionnaire instead of other techniques such as interview sessions and usability lab testing. First and foremost it is a very cheap method, both in the sense of money (compared to renting a usability lab) and time (conducting several interviews and performing usability testing). While it is possible to motivate arbitrary people to test the games, it may be very hard to convince them to do longer usability testing or interviews, especially without some serious incentives. The results of the survey

and observations done during playtests will serve as the basis for evaluating the games and multiplayer modes.

CHAPTER

4

TEST ENVIRONMENT

This chapter will detail what devices will be used for the testing and development of Flash games for MOOSES.

4.1 Desktop computer

The desktop computer has mainly been used for graphics manipulation as well as general developing and report writing. It is equipped with an Intel Core2Duo 2.4 GHz processor, 4GB RAM and a ATI Radeon 4850HD graphics card with 512MB memory. The operating system installed is Windows 7 64-bit.

4.2 Laptop computer

The laptop computer has mainly been used for general developing and report writing when away from the desktop computer. It is equipped with an Intel Pentium Dual 1.87 GHz processor, 2GB RAM and an Intel based graphics card. The operating system installed is Windows 7 32-bit.

4.3 Television

The author has a 42 inch LCD television available supporting resolution up to 1920 times 1080 pixels. This will be used for testing MOOSES games.

4.4 Mobile phone

The author has both a Sony Ericsson K800i and a HTC Magic. Only the K800i supports the MOOSES client, and therefore it will be used for testing. In addition, Alf Inge Wang has provided seven Sony Ericsson phones for testing purposes.

CHAPTER

5

DEVELOPMENT TOOLS AND SOFTWARE

This chapter will detail what tools and software will be used for the testing and development of Flash games for MOOSES.

5.1 FlashDevelop

FlashDevelop is an open source, source code editor for Windows supporting the Flash scripting language ActionScript. It has code completion and syntax highlighting for web related languages like PHP, HTML, XML and CSS as well. It features a look and feel that is similar to Eclipse and Visual Studio, as seen in Figure 5.1. The interface is intuitive and simple. Unfortunately, it lacks debugging tools like pausing and stepping, and inspecting variables. Despite this, it is probably the best free alternative for developing Flash components and games.

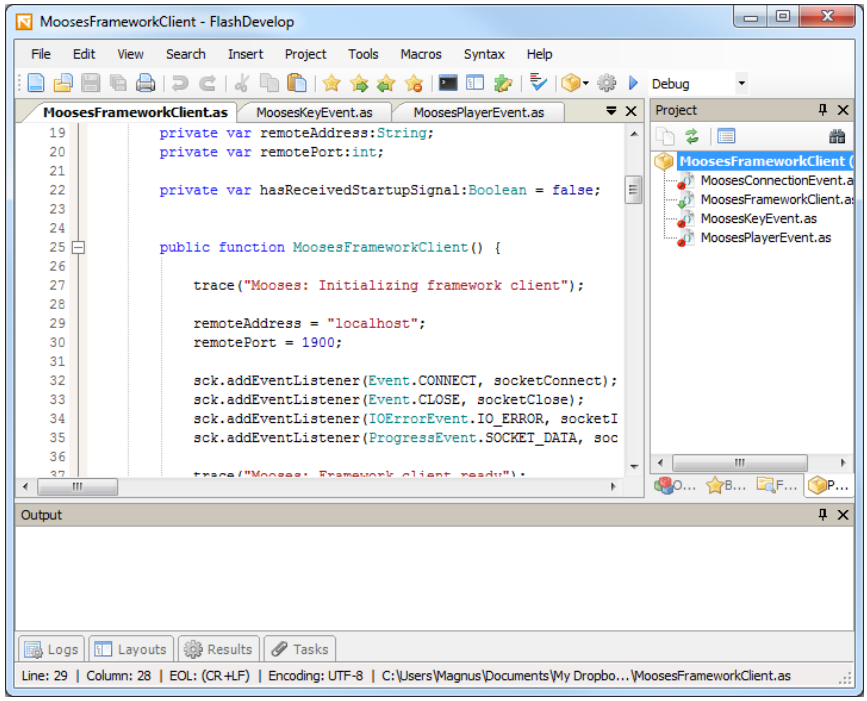


Figure 5.1: *FlashDevelop workspace.*

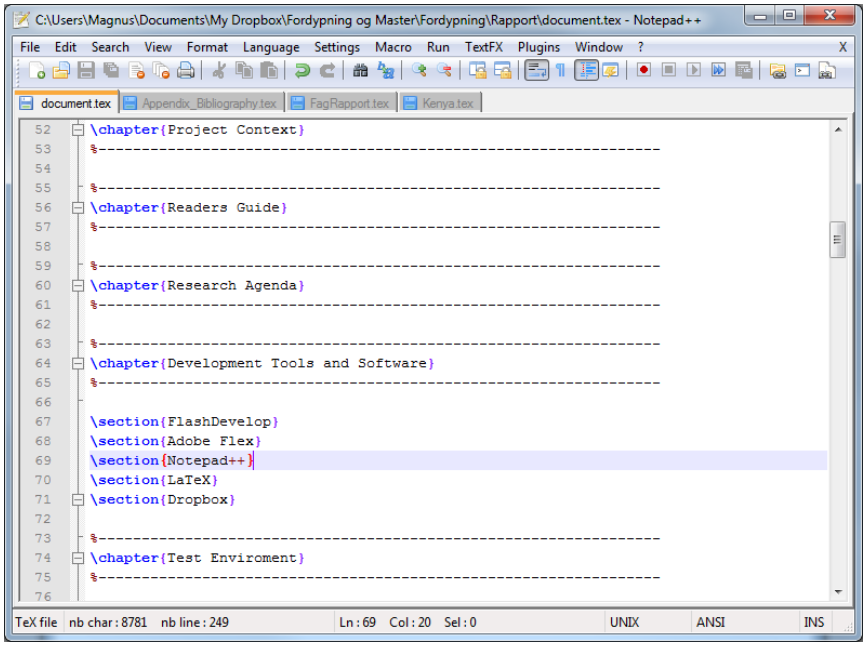


Figure 5.2: *Notepad++ workspace.*

5.2 Notepad++

Notepad++ is a free alternative to Windows Notepad with syntax highlighting for a great number of languages. Other features are plugins, macros as well as a tabbed interface, see Figure 5.2. It is also possible to split the view of one file into two distinct parts, so that one can edit different areas of one file at the same time in the same view. This program is used to write the report as well as edit XML-files and other text files.

5.3 Paint .NET

Paint .NET is a free graphics manipulation software used to create game graphics and report figures. The interface is seen in Figure 5.3.

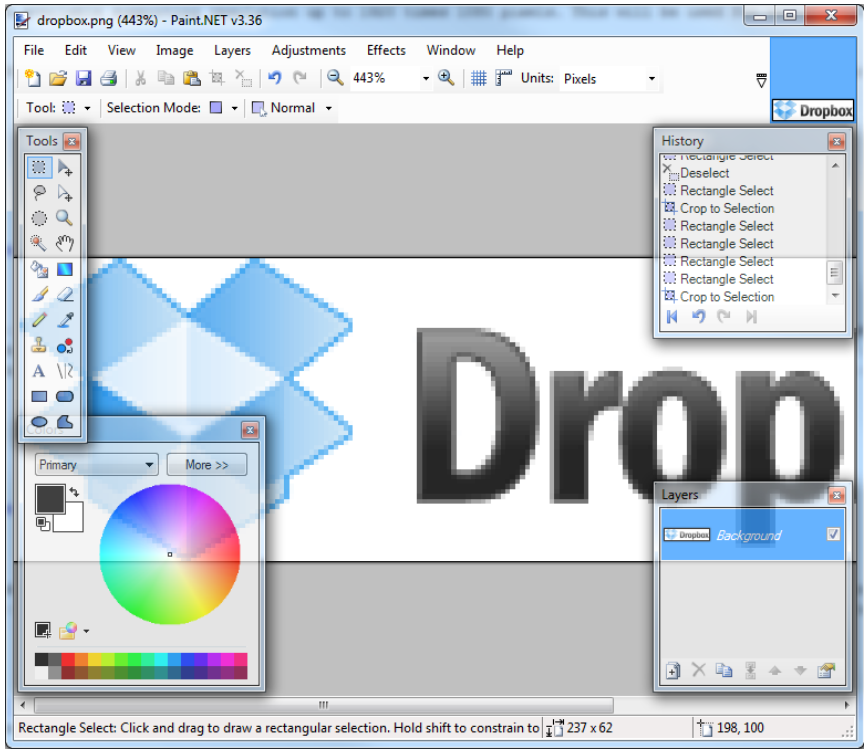


Figure 5.3: *Paint .NET workspace.*

Figure 5.4: *Dropbox logo.*

5.4 Dropbox

Dropbox, logo shown in Figure 5.4, is a personal backup and versioning tool which automatically synchronizes a folder on your computer with an internet server. Older versions as well as backups can be restored, and files are accessible from anywhere by logging in to the Dropbox site. The entire project as well as the report is placed within the Dropbox folder to make sure everything is backed up and available.

5.5 Adobe Flex

According to Wikipedia [Wikb], "Adobe Flex is a software development kit released by Adobe Systems for the development and deployment of cross-platform rich internet applications based on the Adobe Flash platform". Flex allows a programmer to approach building a Flash application more like a software product, opposed to the traditional animation approach. User interfaces are defined by using XML-files, and behavior and actions are implemented in ActionScript. Included is also a freely available compiler from Adobe, which allows a programmer to use nothing more than a text editor to create Flash programs and animations.

5.6 MikTex

MikTex is a LaTeX-distribution for Windows used for creating this report.

Part III

Prestudy

CHAPTER

6

VIDEO GAMES

According to Wikipedia, a video game is "... an electronic game that involves interaction with a user interface to generate visual feedback on a video device" [Wikc]. The system which the games run on is called a platform, and can be anything from a portable device such as the Nintendo DS to a personal computer.

To control the game, an input unit or device is needed. The shape and layout of these devices vary from platform to platform. On handheld devices, the input device is incorporated into the platform. An input device does not have to be just buttons; microphone, accelerometers and even video recognition can be used to control games.

To provide feedback for the player, video games are not limited to only a video screen, but can also use audio, vibration, and force feedback.

6.1 Genres

Computer games are diverse in all aspects of their qualities, from overall theme, to graphics, to audio, and to methods. For this reason, video games are typically categorized in genres by gameplay mechanics, rather than theme and emotional purpose. A strategy game with a western theme has more in common with a

science fiction strategy game than a shooter placed in a western setting. What follows is a (incomplete) list of common genres. It is important to remember that these are just major genres, and games within a genre may be very different in technology or theme. Each genre typically has many sub genres.

6.1.1 Action

The action genre is a genre where games are fast paced, and requires quick reflexes and precision to overcome the obstacles given by the game. While movement and placing are important aspects of this genre, it is usually combat which is the main focus. Popular subgenres are shooters, where projectile combat is the main focus, and platform games, where precise movement of the player is the key to success. Call of Duty 4 (Figure 6.1) and Mario (Figure 6.2) are typical games from each respective subgenre.

Figure 6.1: *Call of Duty 4, first person action shooter game.*

6.1.2 Adventure

As a contrast to the action genre, the adventure games feature little to no action. The player is left to explore the world in his or hers own pace. Obstacles presented are usually puzzles that must be solved in order to advance the plot.

Figure 6.2: *Mario, a platform action game.*

6.1.3 Construction and management simulation

Construction and management simulation games let the player manage resources and/or construct fictional structures in order to achieve and maintain a sustainable resource economy. Games found in this genre are city-building games (popular example is SimCity, see Figure 6.3) and business simulation games (such as Theme Park and Transport Tycoon).

Figure 6.3: *Sim City 2000, a city-building game.*

6.1.4 Role-playing

Role-playing games let the player control a character which can specialize in certain skills, such as magic, fighting or bow mastery, and is left to explore a world. The player typically gains experience by completing objectives and, depending on the game, can use the new experience to further improve their skills.

6.1.5 Strategy

A typical strategy game let the player control an army of units, and is to place and direct them in order to win over one or more opponents, or to complete defined

objectives. The size of the army, the type of units, and world to play in is up to each game. The player must carefully plan the movements his army must take in order to win. Examples of strategy games are Starcraft (see Figure 6.4), Red Alert series (see Figure 6.5) and chess.

Figure 6.4: *Starcraft, a real-time strategy game.*

6.1.6 Vehicle simulation

Vehicle simulation games attempt to let the player control various vehicles and simulate these vehicles as accurately as possible. This genre usually benefits the most from addition peripherals such as electronic steering wheels.

6.2 Multiplayer

A game is said to support multiplayer if two or more players can interact in the same game session. Multiplayer can be achieved by using different methods, where some of them will be detailed in this section.

Figure 6.5: *Red Alert 2, a real-time strategy game.*

6.2.1 "Hot seat"

Hot seat is a method where the players share the same screen and take turn controlling the game with only one input unit. This method typically works well for turn based games, such as Worms and Civilization.

6.2.2 Network

The most used method of allowing multiplayer is by letting each player have their own screen and input, and synchronize the game session by using a network connection. Some turn-based strategy games offer the possibility of playing by email.

6.2.3 Split-screen

Split-screen is a method of multiplayer where the players share the same screen, but have individual input units.

There are many different methods for having several players sharing the same screen. One can assign a portion of the screen to each player, commonly known

Figure 6.6: *MotoGP 3 in split-screen mode.*

as split-screen (Figure 6.6), or the screen can show the entire playing field at once, including the players. Variation of the latter can be to zoom and crop the view to only display the parts of the playfield interesting to the players, such as in football games where the camera focus on the ball. While no longer technically split-screen, the principles are the same.

Split-screen is a very common method used by video games to give players the possibility of playing together without using network connectivity or taking turns. MOOSES use the split-screen principle, one screen with multiple input units, however if the actual screen should be split is a design decision left to the MOOSES game developers.

6.3 Multiplayer interaction

There are many ways of letting the players play together, as detailed in the previous section. There are also many ways the players can interact with each other. This section will try to expand on some of them.

6.3.1 Team versus team

Players can be divided into teams, and then be encouraged to play in the best interest of the team. A sense of camaraderie and us versus them often motivates players, and cooperation is the key to success. How the players interact within a team and with the other team may vary greatly. The players can cooperate by controlling one game character each, or in some way share control of the team entity. A football game may give each player control of one football player, and play against another team. A strategy game could have multiple players on each team, where each player can issue commands and thus have equal control of the team entity.

6.3.2 Player versus player

Players are only encouraged to promote their own interests, which may or may not be in the best interest of other players.

CHAPTER

7

MOOSES

The first implementation of the MOOSES (Multiplayer On One Screen Entertainment System) framework were developed as a part of the students Sverre Morka, Aleksander Baumann Spro and Morten Versvik depth-study in the autumn of 2006. It has later been developed and maintained in cooperation with Tellu AS, a technology company located in Asker. The purpose of the framework is to easily create games for large screen using the mobile phone as a controller.

7.1 Design and architecture

MOOSES is implemented using Java technology, requiring support for J2SE on the server side and J2ME on the client side. Although most mobile phones support J2ME these days, new and popular phones such as the iPhone and Android-based phones do not. In time, the client side will be made technology independent.

7.1.1 High level architecture

The MOOSES framework is designed to be modular, in order to allow future additions and modifications. The high level system design consists of a communication server, a user agent, a game server, the game, a login module , a billing module,

a client handler, a framework client and a game client. Figure 7.1 shows the communication between the various modules.

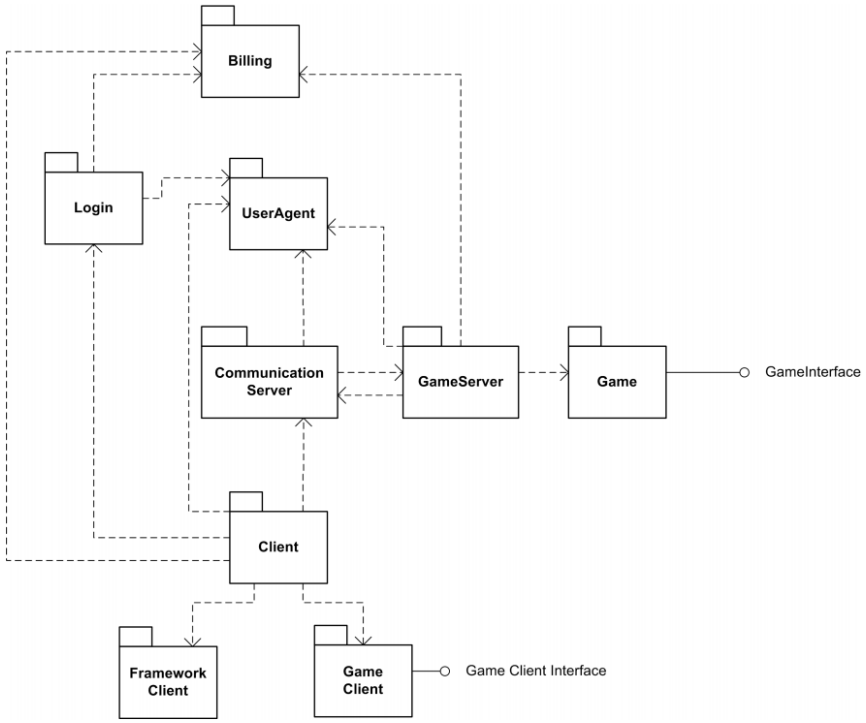


Figure 7.1: High level view of the MOOSES framework.

Billing Handles the user billing.

Client Handling the client requests and framework feedback.

CommunicationServer Provides access for to the framework for the clients. It also is responsible to create the initial connection between all modules.

FrameworkClient Handles billing, login and other requests related to the framework.

Game The game module is independent from the framework, but need to support the given interface.

GameClient Is linked to a specific game, which makes it possible to tailor the game controllers.

GameServer Handles the information between the framework and the game, and is responsible for loading game modules.

Login Authenticates the users when they tries to connect to the framework

UserAgent Contains all the information to a user, and the communication information to- and from the user.

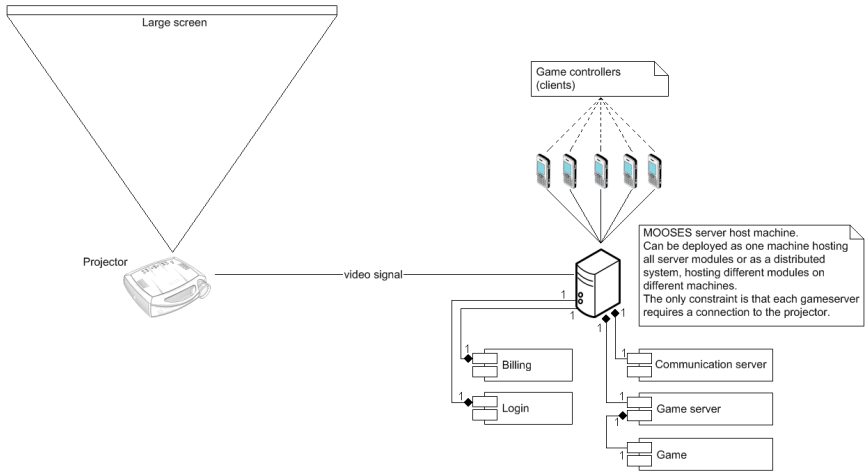


Figure 7.2: Physical view of the MOOSES framework.

7.2 MOOSES and Bluetooth

The MOOSES framework use Bluetooth technology to communicate with the users mobile phones. Bluetooth is a short range radio data communication protocol intended for a wide array of devices, and almost every phone today supports the standard. Popular uses are wireless connection between devices such as mouse and keyboard and a computer or other device. Wireless headsets for phones are also a popular use. The standard is designed to use little power and the chipsets it is based on are low cost.

MOOSES utilizes a Bluetooth hub which supports up to 21 connections per access point, and more access points can be set up without modifying the MOOSES framework. MOOSES also supports regular Bluetooth devices, which only allows up to seven connections. Although not very impressive, it is very useful during testing to do not have to handle the extra hardware.

CHAPTER

8

ADOBE FLASH

8.1 What is Adobe Flash

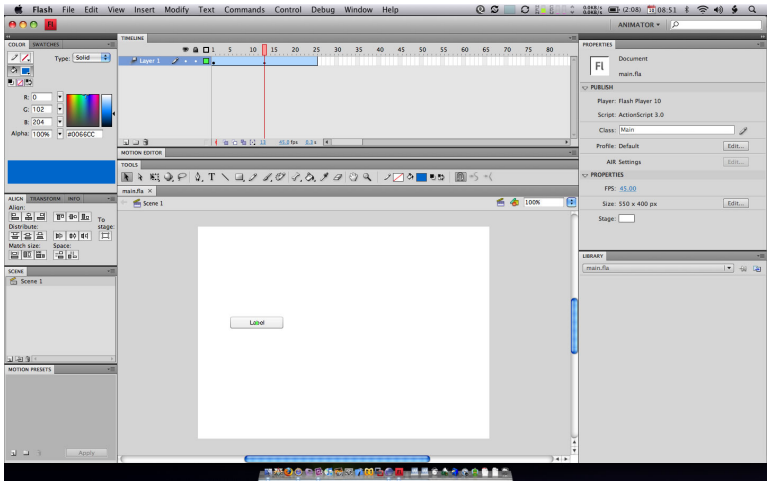


Figure 8.1: Adobe Flash CS4 Professional.

Adobe Flash (previously Macromedia Flash) is a multimedia platform maintained and developed by Adobe Systems [Wika]. It has become a very popular technology

to incorporate animation, interactivity and video to web pages. Most common uses is to create animated advertisement, rich web applications and components, and relatively simple browser games.

Flash has grown over time, from vector and raster graphics manipulation to supporting bidirectional video and audio streaming. Scripting is enabled by supporting a scripting language called ActionScript.

To view Flash movies and components, one needs to have Flash Player installed, either as a stand-alone player or as a plugin to a browser. Adobe Flash Player has been implemented on all the major operating systems, as well as a light version for mobile devices.

8.1.1 Popular uses of Flash

According Adobes web page [Ado], almost 99% of computers connected to the internet can view content based on Flash, and has almost become a necessity to view the modern web. Several web pages have been built on using Flash technology, such as YouTube, DailyMotion, Armor Games, and some web pages are built entirely of Flash.

Advertisment Flash allows animation, music and video which is used extensively to catch the users eye and create attractive commercial elements.

Web components Flash can be used and is used to replace otherwise standard HTML web components such as menus, introduction screens or even a complete web page.

Prototyping Flash offers a familiar scripting language (ActionScript3) and allows for rapid testing of ideas. The amount of work needed to get an idea moving on the screen is very low.

Video embedding Flash can be used to embed video in web pages. Popular sites like YouTube and DailyMotion allows users to upload their videos and share them with the world.

Games Flash has become a very popular game creation platform. The games are often so-called casual games, intended to be played for a limited time on an irregular basis. Web sites such as Armor Games hosts a large number of free games made in Flash.

8.1.2 ActionScript

In order to create interactive animations and components, Flash was given support for a scripting language, ActionScript. This language has evolved a lot. First implementation was very simple, only supporting basic navigation commands such as Play, Stop, GoToFrame and GoToUrl. The current version of the scripting language is called ActionScript 3, and is a full-fledged object oriented programming language. The latest version of Flash Player now includes a Just-In-Time compiler, which increases the performance of Flash significantly.

Although intended as scripting language for Adobe Flash's authoring tool, it can be used by itself to create Flash files using new tools released by Adobe. This allows a developer to create a game or an application without using anything but a text editor and a compiler.

```
package com.example
{
  import flash.text.TextField;
  import flash.display.Sprite;

  public class Greeter extends Sprite
  {
    public function Greeter()
    {
      var txtHello:TextField = new TextField();
      txtHello.text = "Hello World";
      addChild(txtHello);
    }
  }
}
```

A Hello World example in ActionScript 3. Notice how variables are declared differently from Java and C, with the type after the name of the variable.

8.2 Benefits of Adobe Flash for MOOSES

There are two major benefits of using Adobe Flash in conjunction with MOOSES. The first is prototyping. It is very easy to create prototype games in Flash, and allowing prototype games to be tested on the large screen with actual hardware is a huge benefit when testing new gameplay elements.

A second benefit is that Flash is used by more than just programmers; advertising

studios might have expertise in using Flash, but not so much in more classical programming environments. Allowing Flash makes the MOOSES platform available for highly skilled persons within advertising. One could imagine such persons create a mini-game to advertise a product, and this mini-game could be played before a movie starts as an interactive commercial.

CHAPTER

9

STATE OF THE ART

9.1 Previous work at NTNU

In their master's thesis, Spro and Versvik [SV07] take a look at the social attributes of gaming in general and especially of MOOSES games. They also evaluate and suggest improvements to the MOOSES framework and implemented MOOSES games.

Kvasbø, in his master's thesis [Kva07], evaluated several different game concepts which could be feasible for the MOOSES framework.

Morka [Mor07] developed a scriptable client for the MOOSES framework. This allows a customization of the client without redistribution of a recompiled version.

Heggdal [Heg08] explored the possibility of allowing games based on XNA (a game framework implemented in C#) to connect to the framework. This required the framework to support a different communication channel than JNI (Java Native Interface) which C# does not support. The communication method chosen was a standard TCP-connection, allowing a number of new technologies to interact with the framework.

Føllesdal [Føl09] implemented and evaluated a game called Selfish for the MOOSES framework.

9.2 Multiplayer On One Screen Games

This section describes some systems which allow players to interact using a single, shared screen.

9.2.1 ProjectorGames

Figure 9.1: A ProjectorGames event.

ProjectorGames is a company who specializes in hosting game sessions using a projector, custom controllers and custom games fitted for the large screen. They develop their own games and use them for large screen events and publish them to Xbox Live. According to their website [Pro], their large screen system can be used by up 512 players at once. All in all, their concept is very similar to MOOSES however ProjectorGames have to supply a controller to all the players, unlike MOOSES where they can use their own mobile phones and other devices.

They have developed many interesting and promising game concepts which could be transferrable to MOOSES and should probably be studied further.

9.2.2 Carry Small, Game Large

In an article called "Carry Small, Game Large: Big Shared Screen Multiplayer Gaming"[RJCW], published on a popular game development website called Gamasutra, a new kind of game was presented. The players must, as with MOOSES, be in the same physical location and share the same, large screen. Unlike MOOSES, the clients can by any device capable of processing JavaScript techniques such as

Figure 9.2: Carry Small, Game Large. Left half is the client screen, the right half is the screen shared by all players.

AJAX. Since the client is HTML/JavaScript based, it does not require any installation, players simply have to go to a shared URL and log in.

This approach may be satisfactory for many games that do not have strict latency requirements, but for fast-paced games the latency might be an issue.

9.3 Flash Games

Typical Flash games today are intended for the casual audience, players who have a limited playtime and just want some instant entertainment. These games have a very limited set of features, simple graphics and addictive gameplay. Example of such a game is Bloons, seen in Figure 9.3, where the main objective is to throw a limited number of darts in order to pop as many balloons as possible. Another very popular type of Flash game is tower defense. In tower defense games, such as GemCraft seen in Figure 9.4, the player must place defensive structures along a route where enemies will pass through and attempt to reach the end of their path.

There are not just simple games in Flash. Machinarium is an adventure game where you control a very special robot. The game has a very unique and beautiful art style, and this is enforced in the game by using little or no text. A screenshot from the game is seen in Figure 9.5.

The "weird" and experimental gameplay is also typical for Flash games. Flash is easy and intuitive to begin with, and is used by a great number of people with different backgrounds, from artists to advertisers to programmers. It is often used as a prototyping platform.

There are some issues with Flash. The performance is not great as it does not utilize modern graphics cards. Although there are attempts of creating 3D games

using Flash, these are often very simple compared to the regular PC standard, due to the lack of processing power and lack of official 3D rendering support.

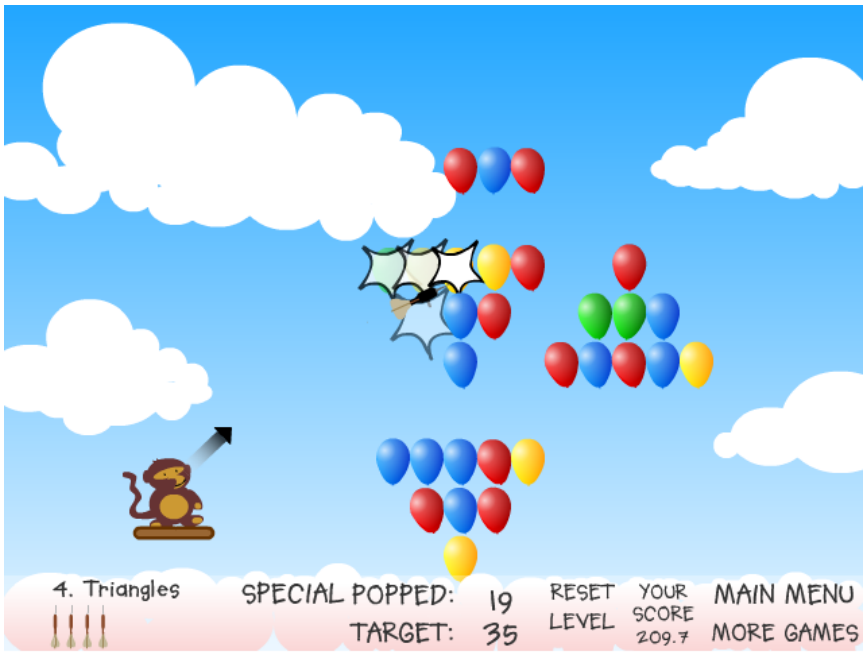


Figure 9.3: *Bloons, a popular flash game [Nin]. The goal is simply to pop a minimum amount of balloons per level.*

Figure 9.4: *GemCraft, a flash game in the tower defense genre [Arm].*

Figure 9.5: *Machinarium, an adventure flash game with a very special art style [Des].*

Part IV

Flash For MOOSES Framework

CHAPTER

10

FLASH FOR MOOSES FRAMEWORK

10.1 Overview

One aspect of the project is to facilitate the creation of games for MOOSES using Flash. A similar challenge was faced by Vebjørn Heggdal when he was to use C# and XNA to create a game for MOOSES [Heg08]. To solve this, the maintainers of MOOSES created a TCP connection which games could connect to. This TCP connection is language and technology independent and requires just an understanding of the protocol MOOSES use. In contrast to previous games for MOOSES, games based on this TCP connection runs in a different process which helps the stability of MOOSES. If a game crashes it will no longer crash MOOSES with it.

To aid the development of games and other applications for MOOSES in Flash, a collection of classes and helper applications, named Flash for MOOSES Framework, was developed. These classes were designed to be as generic as possible in regards to use (no ties to a specific game type) and MOOSES client, while still being as extensive as possible to help development. These classes will be detailed and explained in the following sections.

10.2 Flash For MOOSES Classes

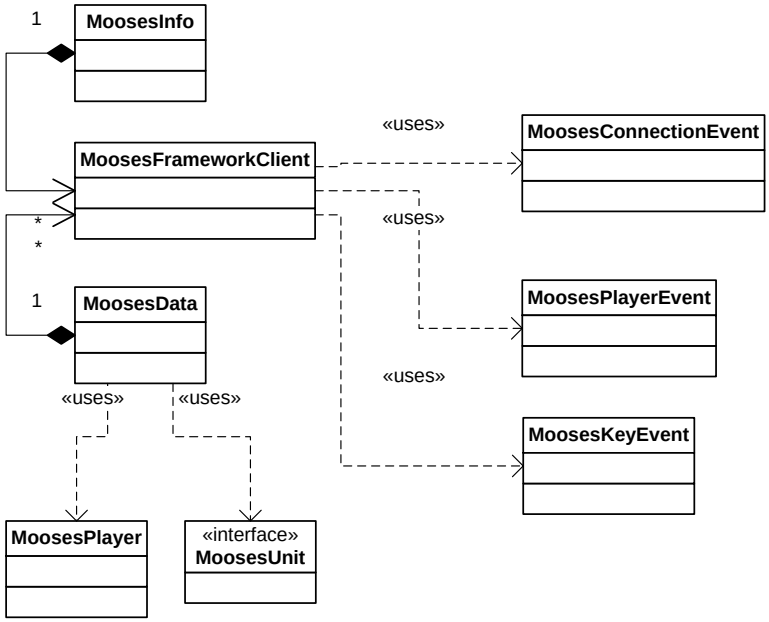


Figure 10.1: *The Flash for MOOSES classes.*

The FMF consists of three major and separate classes: MoosesFrameworkClient, MoosesData and MoosesInfo. Each of them has a different responsibility and utilizes some helper classes and interfaces. The relationship is shown in Figure 10.1. The FMF classes as a whole maintain the connection with MOOSES and propagates events regarding the connection status, player events and key presses.

10.2.1 MoosesFrameworkClient

The MoosesFrameworkClient class maintains a connection with MOOSES and listens to incoming messages as well as sending responses. Events are generated here depending on messages recieved and changes to the connection. These events are picked up by the two other classes, MoosesData and MoosesInfo.

MOOSES message protocol

The structure of messages sent by MOOSES varies by the type of message sent, but the header of the messages always remain the same. The header contains two

integer numbers, first a message type number and secondly the size of the message in bytes, excluding the header.

MOOSES allows games to send data to the client. How this data is treated is up to the client. The message structure to send client messages is as follows: a message type number (integer), played identification number (integer), the number of data units to send (byte). After this header each unit of data is sent. A unit of data is sent by sending a data unit type number (byte), then the data. To illustrate the message structure, suppose one would want to send the number 42 to a player number 10 with the message type 7. The message will then be i7-i10-b1-b2-i42, where i in front of the number indicates integer and b indicates byte. The data unit type number for integer is set to 2.

In order to let MOOSES know what score a player has, a special message can be sent. The structure of this message is first the message type number (1030 for score update), the player identification number and finally the score value.

Events fired by MoosesFrameworkClient

MoosesConnectionEvent This type of event is fired whenever the status of the connection with MOOSES changes. It can be in one of four states, MOOSES_CONNECTED, MOOSES_ATTEMPTCONNECT, MOOSES_ERRORCONNEC or MOOSES_DISCONNECTED.

MoosesPlayerEvent When the status of the players change this type of event is dispatched. It can be in one of two states, either MOOSES_PLAYER_JOIN or MOOSES_PLAYER_QUIT. A unique number generated by MOOSES identifying the player is always included in the event, however the name of the player is only present when a player joins. If used, the MoosesData module will process this event and keep an updated list of current players.

MoosesKeyEvent If a player press or release a key on their MOOSES controller this event is generated. The event will contain the data of which player it concerns, which key and what state the key is in (either pressed, released or clicked).

The other co-modules of FMF and games who would want to use FMF will listen to MoosesFrameworkClient and process the events as they see fit. While the co-modules is not strictly necessary to create a proper MOOSES game, as shown in the next sections, they can be very helpful.

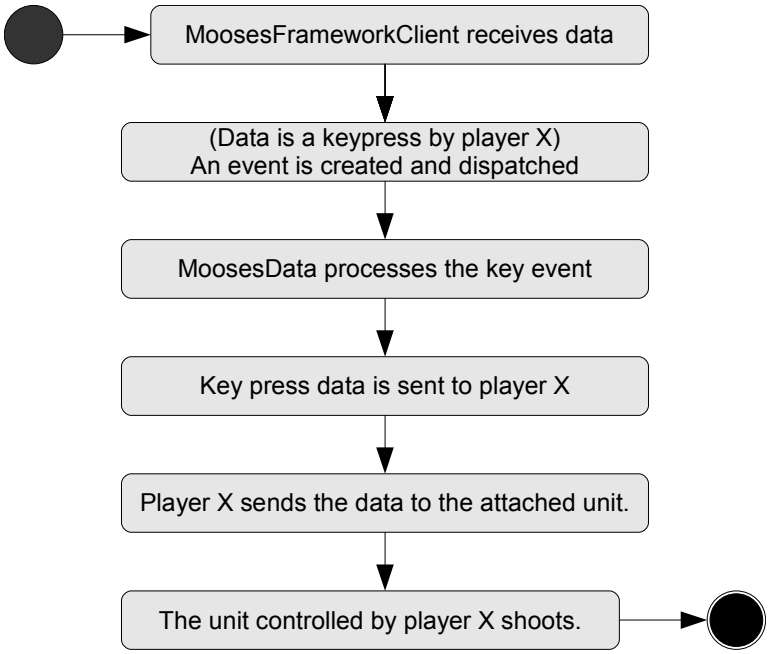


Figure 10.2: *A simple scenario which attempts to show how an event propagates through FMF. While it may seem like an unnecessary amount of steps, this allows for a good amount of freedom for the game creator and it keeps the FMF distinct from the game and highly reusable.*

10.2.2 MoosesData

The MoosesData module is an optional (although very useful) helper module which is responsible for keeping record of connected players, generic player related events and data. The list of which players who are connected is kept up-to-date by listening to the MoosesFrameworkClient.

When a player joins the server, a MoosesPlayer object is created and pushed to the list of connected players. This object contains the player identification number and the player name, as well as some key functions a MOOSES game can benefit from. This is the abstract representation of the player in FMF, as it does not appear graphically, and is distinct from a game unit controlled by a player.

A MOOSES game can iterate through the list of players, and attach a player controlled unit to a player. The player controlled unit must implement the MoosesUnit interface. When a MoosesKeyEvent is picked up by the MoosesData, it will send the data to the correct player, and, if a unit is attached, pass the data along to the unit. In this way, a game unit will get direct input from the players without having to poll for changes in the input status. This also keeps the game implementation separate from the reusable components.

MoosesData can buzz (or vibrate a user's phone, however this depends on the controller) a player based on the unit, freeing the game from having to look up which player identification each game unit has. Instead the game can send buzzPlayer with the game unit to buzz to MoosesData, and the module will go through the list of connected players and send the proper identification to MoosesFrameworkClient.

10.2.3 MoosesInfo

MoosesInfo is an optional module which logs the events fired by the MoosesFrameworkClient. The events are logged by sending it to trace (similar to using System.out.print() in Java) and by storing a string representation of the event in a TextField. Custom messages added by the game creator can also be logged by MoosesInfo. MoosesInfo can be added to a DisplayContainer in Flash to visually see the event log as well as see the status of the MOOSES connection. This module is useful for debugging, and should probably not be viewed by a regular user of the system.

10.3 Helper programs

In order to run Flash games and applications successfully two helper programs were created.

10.3.1 Cross-site Scripting Helper

As a security measure, the Flash runtime does not allow Flash programs to create a TCP connection unless the connected party responds with a security token. This is to combat a security vulnerability called cross-site scripting. The security token is a XML string which detail the rights the Flash program is given. The token is intercepted and processed by the Flash runtime without the Flash program ever seeing it.

The token can be transmitted over the same connection the Flash program is trying to initiate, or it can be sent over a different TCP connection which is initiated by the Flash runtime. Since the TCP connection MOOSES provides is supposed to be language and technology independent, modifying it to send the security token would be a poor idea. Instead, a simple helper program was created. This program waits for an incoming connection, hands out the security token, closes the connection and begins waiting again. The Flash runtime will contact this program and the connection initiated by the Flash game will be allowed.

10.3.2 Flash Game Launcher

External games (games not bundled with MOOSES) must exist in a specific folder in the MOOSES system. The external games must also be executable files. Flash games are distributed as SWF-files, which in turn must be launched a Flash player. While MOOSES could have been modified to recognize the SWF-files and launch the appropriate Flash player, a different solution was found.

A simple helper program was created which simply launches the Flash player and tells it to play the SWF-file with the same file name as the program. As an example, if you have a Flash game named “achtung.swf”, one would rename the program to “achtung.exe” and put the program in the same folder as “achtung.swf”. This way, the program does not have to be recompiled for each new game created, and MOOSES does not have to be modified.

Part V

Game Development

CHAPTER

11

INTRODUCTION

In order to test developing with the created framework and to test and evaluate Flash MOOSES games three games were created. Each game will try to explore different ways the players can play and work together. Tower Defense puts the players on opposing team, and each player controls one character. ProductBall also put the players on opposing teams, but they need to collaborate on controlling their team character. Both of those games should require teamwork and communication for the team to succeed. The last game, Achtung, pits the players against each other in a game of survival. This part will describe the concept of each game in detail, as well as how they were implemented. In the next part, each game will be evaluated.

CHAPTER

12

TOWERDEFENSE

12.1 Concept

This game was inspired by popular tower defense games seen all over the internet and on handheld devices today. A tower defense game is a game where the player must strategically place defensive towers on the battlefield to defend against an incoming horde of enemies. The towers may have different abilities, strengths and weaknesses. The enemies may follow a pre-determined path, or be forced to follow a certain path because of the defensive structures placed by the player, or even just roam freely around the battlefield. The main objective of the game is to stop the enemies to reach a certain point, the other side of the battlefield, or the end of a path.

To adapt this game concept to the MOOSES several adjustments will be made. The MOOSES players will be split into two teams, attackers and defenders. The attackers must reach the opposite side of the screen, the defenders must stop them. The attackers may run freely around the battlefield, while the defenders are stationary, see Figure 12.3. The game should have three distinct states, a deployment state where the defenders place their towers, a game state where the attackers attempt to reach the other side, and an intermediary state where statistics and information could be displayed, and new players are allowed to join.

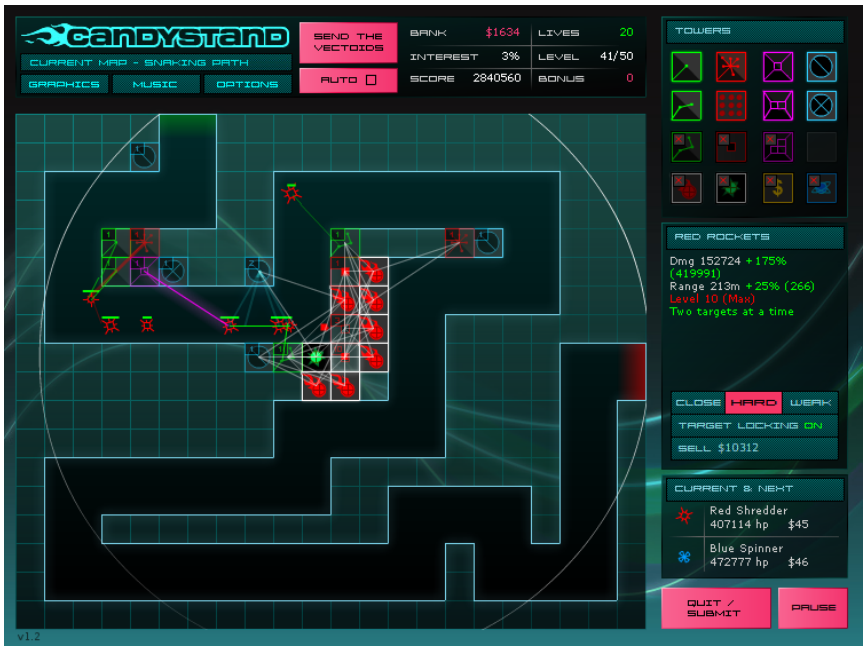


Figure 12.1: *Vector Tower Defense.*

Figure 12.2: *Omega Tower Defense.*

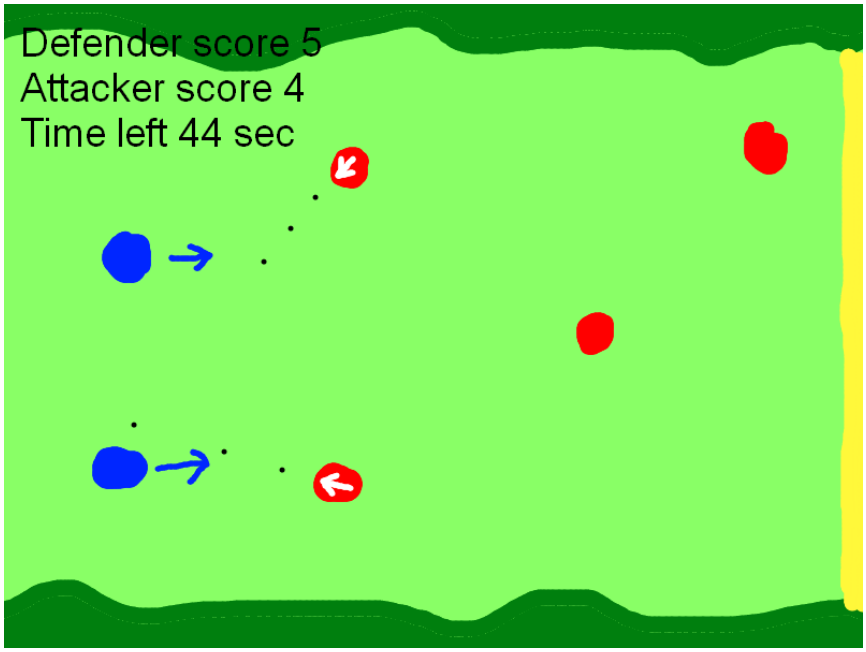


Figure 12.3: *Illustration of the game concept. The attackers (in blue) attempt to get to the other end of the playfield (the yellow line on the left) and the defenders (in red) tries to prevent this by aiming and shooting the attackers.*

The multiplayer mode will therefore be team versus team, but each player controls one character. Hopefully the game concept should allow and promote team play. Attackers should want to work together and apply various tactics to reach the other side, while the defenders must place their towers strategically in relation to the other defenders. To "spice up" the gameplay, different power-ups (items which gives temporary advantages to a player) can be placed on the battlefield, either for the attackers to pick up, or for the defenders to shoot. The battlefield may also contain items which blocks the path of both attackers and defender bullets.

To balance the game it may be beneficial to have a different number of players on the attacker and defender side, with more players on the attacker side. A good ratio should be found by testing. When an attacker reaches the other side the attackers score a point, when a defender kills an attacker they score a point. The game will end after a certain amount of points for either side. If an attacker reach the other side or is killed by a defender it will spawn again at the beginning. There will be a maximum number of respawns, and if the attackers have not reached their goal by the time the round is over the defenders will win.

12.2 Implementation

This section will introduce TowerDefense and explain how the various bits and pieces were implemented. A screenshot from the final game can be seen in Figure 12.4. A class diagram of the final system can be seen in Figure 12.5.

Figure 12.4: *The TowerDefense game.*

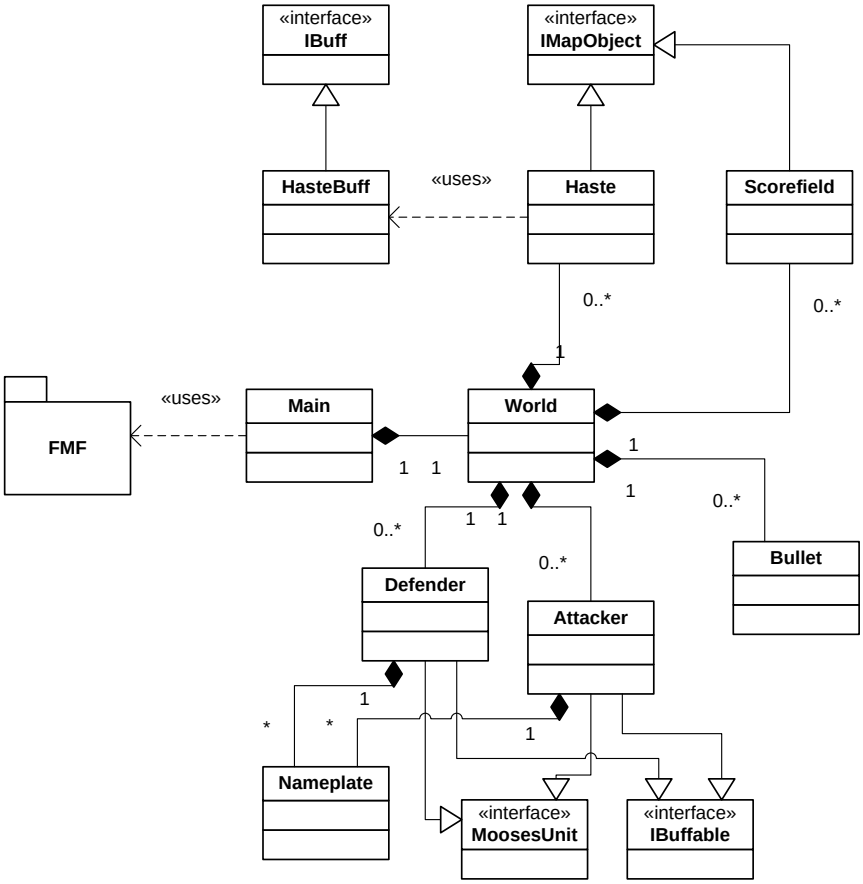


Figure 12.5: *TowerDefense: class diagram.*

12.2.1 Game states

The game moves between four distinct states, a limbo state, a deployment state, an battle state, a game over state, all which will be described in detail in the following sections.

Limbo

The game starts in a *limbo* state. The purpose of this state is to allow the MOOSES framework to send the list of connected players and to allow players who connect after the game was launched to join in. This is a pause state, and players cannot move any units.

Deployment

In this state, the defenders are allow to move about and choose where to place their tower, see Figure 12.6. They must work together in order make sure they cover all entrances to the attackers score area, represented as star symbols in the game. The attackers are not allowed to move in this state, but are encouraged to try to devise a plan to breach through the defenders towers.

Figure 12.6: *TowerDefense: deployment state. The red tint near the borders is areas where the defenders are not allowed to place their towers. The white circles show their range. The attackers are lined up on the left side awaiting the next state.*

An important note to make is that when the *deployment* state is initiated, the

connected MOOSES players are assigned either an attacker or a defender. This is done by retrieving the player list and assigning a minimum of one player as a defender up to a certain percent of the connected players. The rest of the players who are not assigned to be a defender will be assigned to be an attacker.

When the game enters the *deployment* state it will also reset the game board, remove all objects, tiles and graphical elements. After the cleanup, necessary objects will be added.

Battle

In the *battle* state the attackers are allowed to move, while the defenders must remain stationary. The defenders are now capable of shooting, and must do so in order to stop the attackers from scoring points, see Figure 12.7.

Figure 12.7: *TowerDefense: battle state. The attackers are attempting to reach the star symbols, their goal. The defenders are shooting at the attackers and trying to stop them.*

The attackers run over haste objects in order to increase their speed, and defenders can shoot haste objects to increase their rate of fire.

This state ends if the attackers no longer can respawn, if enough attackers have breached the defenses, or if time runs out. The side with the best score will be declared the winner.

Game over

When the *battle* state has ended, the game is paused and a score is displayed. If the game is set to span over several rounds, the next state will be *deployment*. If it is set to only be one round, the game will exit. The time spent in this state will allow new players to join the MOOSES server and be a part of the next round as each player is assigned a new game unit in the *deployment* state.

12.2.2 User interface

The user interface of the game can be seen in Figure 12.8, and explained in the list below.

Figure 12.8: *TowerDefense: user interface.*

1. A progress bar indicating the time remaining.

2. A number indicating how many times the attackers have reached the star symbols on the other side (9). When this reaches zero, the attackers have won.

3. A number indicating how many more times the attackers can respawn if killed by the defenders.

4. A haste object which can be picked up by either side. The attackers have to run over it, the defenders must shoot it. Attackers will run faster, and the defenders can shoot faster.

5. Sparkling stars will appear when an attacker picks up a haste object or when the attacker reaches a star symbol.

6. This represents an attacker. The attackers' goal in life is to reach the star symbols and avoid fire from the defenders.

7. The defender, represented by a bunker, tries to stop the attackers by shooting at them.

8. The circle represents the defenders range of fire. This is shown to let the attackers know where not to run, and to let the defenders know how far they can shoot.

9. If an attacker reaches this star symbol they score a point for the team and deducts one life point from the defending side.

12.2.3 Implementation of the World class

The most interesting bits of the game are handled in the World class. It is here connected players are given their unit to control, score is being kept track of, and the visual parts are being handled. To propagate the update game logic message throughout the various elements, the Main class listens for the enter frame event. When this happens it calls the World class' update method. This method will again call the update method of each object handled by the World class.

The playfield

The world is made up of two separate grids lying on top of each other covering the entire screen.

The bottom layer defines mostly the look of the playfield, but can also be queried to find out if a player is allowed to move to a new location. There are five tiles currently implemented (grass, trees, river, bridge and concrete), and each tile has two different boolean properties (traversable, meaning if a player is allowed to walk on it, or blocking, if a bullet can cross this tile).

The upper layer consists of special objects. Two objects exist, a haste power-up, and a score area. The haste power-up will increase the movement speed of attackers who pick it up, and the rate of fire of defenders who shoot it.

Game units

The World class maintains one list of attackers, and one list of defenders. These lists are populated in the beginning of the deployment state by looking at the players connected to MOOSES.

Particles

Simple particle effects are created by having a list of sprites to draw, move and remove after a certain time.

State transition

The various states of the game have been explained in more detail in the previous sections, and the World class will move the game into a new state when the necessary time has elapsed.

User interface

Finally, the World class updates the user interface.

CHAPTER

13

PRODUCTBALL

13.1 Concept

The multiplayer mode in TowerDefense is team versus team, where each player controls their own character. For this game, a different mode will be implemented and tested. The players are going to collaborate on controlling one character per team. The players will need to coordinate with the other players beside them in order to maximize their effectiveness. The hope is that this kind of multiplayer mode will produce a different experience than the one in TowerDefense. While different game concepts may fit this multiplayer mode, the chosen idea was to make something similar to BlobbyVolley seen in Figure 13.1, a volleyball game with blobs.

The game to be created, named ProductBall for the many potential places to put advertising, plays similarly to a volleyball game. The players must attempt to land the ball on the floor on the opponent's side. Instead of punching or throwing the ball, the ball will simply bounce of the head of the players. A mockup of the concept can be seen in Figure 13.2. Each team has one character, and each character is controlled by the players on that team. The character should be easy to control, as the concept of cooperative control might be hard to grasp at first. Therefore, each character can only move left, right or jump. The players must work together to move the character in such a position that the ball will bounce to the opponents

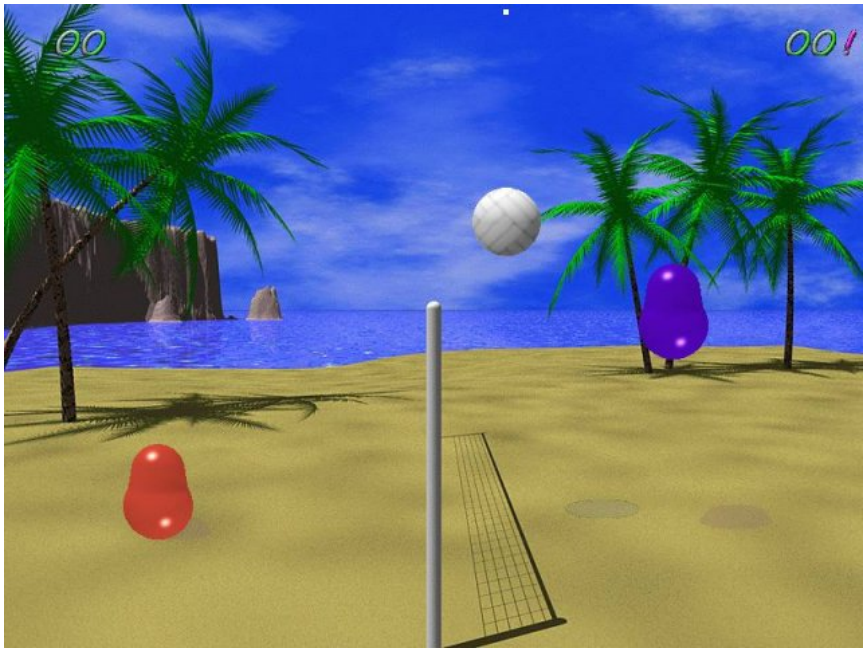


Figure 13.1: *The blobs must jump and position themselves correctly in order to make the ball bounce to the opponents side.*

side.

Figure 13.2: *Each team has one character to control. Ample amount of advertising.*

To control the characters, they player will vote on what the best course of action is at any time. The character will move in the direction the majority of the voters decide, and will move at a greater speed depending how many of the players on that team vote for the direction. To jump, a certain percentage of the players on that team must agree that jumping is the right course of action.

13.2 Implementation

13.2.1 Classes

Figure 13.3 shows the relationship between the different classes and how FMF was used. This subsection will go through each class and briefly detail the responsibility the class has and how it was implemented.

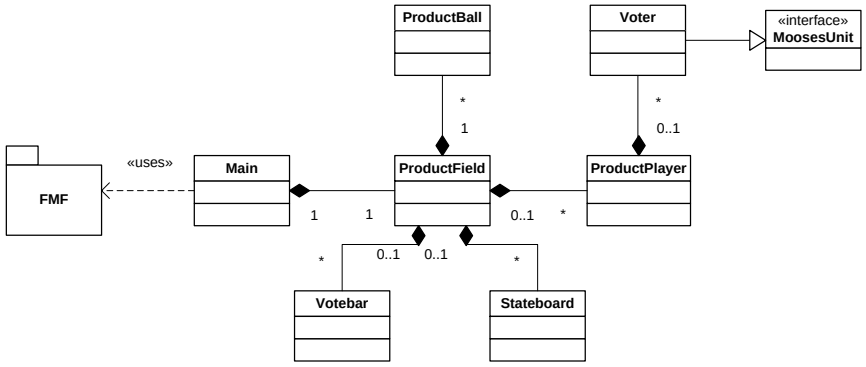


Figure 13.3: *ProductBall: class diagram.*

Main

This is the entry class of the program. When started, the main class will initialize FMF and attempt to connect with MOOSES. Once a connection has been established, the Main class initializes ProductField. The main class also has a timer which sends an update message to ProductField at regular intervals.

ProductField

ProductField is the workhorse of the game and holds the game logic and is ultimately responsible for everything shown on the screen.

When the game is about to begin it divides the connected players into teams. The teams are represented by a ProductPlayer object. The connected players get a Voter object attached to them and the Voter object is added to the correct ProductPlayer.

ProductBall

ProductBall holds the information about the ball and is responsible for making sure that the ball does not go too fast or off the screen.

Votebar

The Votebar is an UI element which displays the amount of players voting for each course of action. Its intention is to give the players some feedback.

Stateboard

The Stateboard is an UI element which can display messages to all the players.

ProductPlayer

ProductPlayer has a list of Voter objects which are a part of its team. At each game tick it will sum up all the votes and take the appropriate action.

Voter

A Voter object is what the player has direct control over. A key input is mapped to each possible vote; idle, left, right and jump. When the game begins the voter objects are added to one ProductPlayer.

13.2.2 States

Limbo

The game starts in a *limbo* state. The purpose of this state is to allow the MOOSES framework to send the list of connected players and to allow players who connect after the game was launched to join in.

GameBeginning

In this state, the players are divided into two teams, left team and right team. Lists with which side the players are on are shown, see Figure 13.4. The way the teams are made up is by attaching a Voter object to each connected player. The Voter objects are placed alternately in the left and right instance of ProductPlayer.

Battle

The two ProductPlayers finds the majority vote by going through the Voter objects they hold. Depending on what the majority votes, it will either move right, left, jump or stand still. The greater the number of people who vote to move in one direction, the faster the player will move. This is as simple as Speed $\bar{}$MaxSpeed

Figure 13.4: *ProductBall: pre-battle state. The players are divided into to teams.*

Figure 13.5: *ProductBall: battle state. The right team is trying to bounce the ball over to the opposite side.*

* (NumVotesInDirection / NumVoters). NumVotesInDirection is number of votes for a specific direction and NumVoters is the number of voters on that team. A screenshot from the battle state can be seen in Figure 13.5.

Score

Figure 13.6: *ProductBall: score state. The left team scored.*

After a team lands a score, there is a brief pause where which team scored is displayed, as seen in Figure 13.6.

GameEnded

After a team reaches the maximum score, there is a brief pause where the match winner is announced before ending the game.

CHAPTER

14

ACHTUNG

14.1 Concept

Two team based multiplayer modes have been created. Achtung will explore all versus all multiplayer mode. The concept is based off an old game called Achtung Die Kurve, see Figure 14.1. Each player is given control of a curve, which grows in the direction chosen by the player. The goal of the game is to avoid growing the curve into an existing curve, and survive the longest. It has similarities with the mobile game Snake and the light cycle racing game in the movie Tron. The players control the direction by either turning left or right. At certain intervals there will be gaps in the curves, see Figure 14.2. These are added to give the players a chance to survive longer and to do evasive maneuvers through other players and themselves.

The intention with this game is to create a multiplayer mode which is very different from the two preceding games. The players will have to battle among themselves to finally produce a winner. Since the original game was a multiplayer on one screen game, the concept did not need any tweaking to work with MOOSES.

Figure 14.1: *The original Achtung Die Kurve. Each player gets one point for each player who dies while they are still alive. This way, the surviving player gets the most points.*

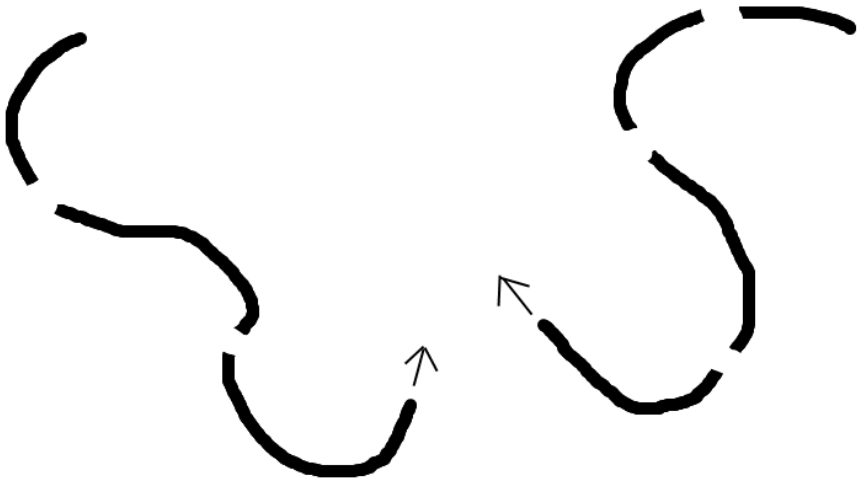


Figure 14.2: *A concept sketch of Achtung. An arrow is placed on the head of each player to show the direction they are going. The players can turn either left or right. Gaps are added at regular intervals.*

14.2 Implementation

This section explains how parts of Achtung were implemented, first by going through the different classes and then by going through the different states of the game.

14.2.1 Classes

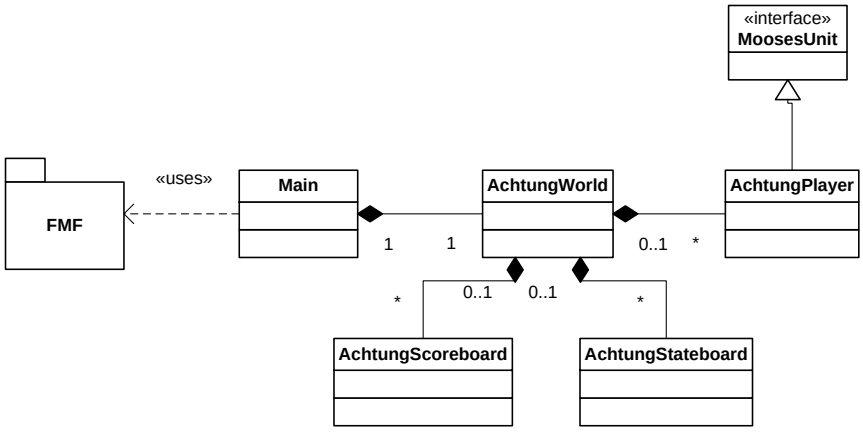


Figure 14.3: *Achtung: class diagram.*

Figure 14.3 shows the relationship between the different classes and how FMF was used. This subsection will go through each class and briefly detail the responsibility the class has and how it was implemented.

Main

This is the entry class of the program. When started, the Main class will initialize FMF and attempt to connect with MOOSES. Once a connection has been established, the Main class initializes AchtungWorld. The Main class also has a timer which sends an update message to AchtungWorld at regular intervals.

AchtungWorld

AchtungWorld holds the game logic and is responsible for everything on the screen.

When a game is about to begin, this class looks at the connected player list and

creates an AchtungPlayer for each of them. The AchtungPlayer is then attached to the connected Players so that input messages are sent to them.

When the main class sends an update message, the AchtungWorld will send an update message to each of the AchtungPlayer objects. Below is a pseudo-code of the update function of AchtungWorld.

```
function update()
{
  for each AchtungPlayer player in playerList {
    player.update(); //move and update direction
    checkForCollision(player);
    draw(player);
  }

  if (numPlayersAlive() < 2) {
    endState();
  }
}
```

If a collision is found then that player is "killed" and all living players are awarded with some points. The one who survive the longest will therefore end up with the highest amount of points that round.

AchtungScoreboard

The AchtungScoreboard is an UI element and shows the score of each player as well as the color of the player.

AchtungStateboard

The AchtungStateboard is an UI element which can display messages to all the players.

AchtungPlayer

AchtungPlayer is the class which holds all the information about the player such as position, heading, name and score. It receives input messages from MOOSES since it is attached to a MoosesPlayer. Based on which keys the player is pressing it adjusts its direction and moves very slightly forward in that direction.

14.2.2 States

This subsection will briefly go through each state in the game.

Limbo

The game starts in a *limbo* state. The purpose of this state is to allow the MOOSES framework to send the list of connected players and to allow players who connect after the game was launched to join in. This is a pause state, and nothing except a message saying "waiting for more players" is shown. After a specified amount of time the game moves to the RoundBeginning state.

RoundBeginning

The game field is now shown, and the players can now look for their character on the screen. A message saying "the game is about to begin" is shown. The players' start positions are arranged in an elliptical pattern and their initial direction is towards the center. This was done in order to give players a fair and consistent start. Figure 14.4 shows the initial pattern quite clearly.

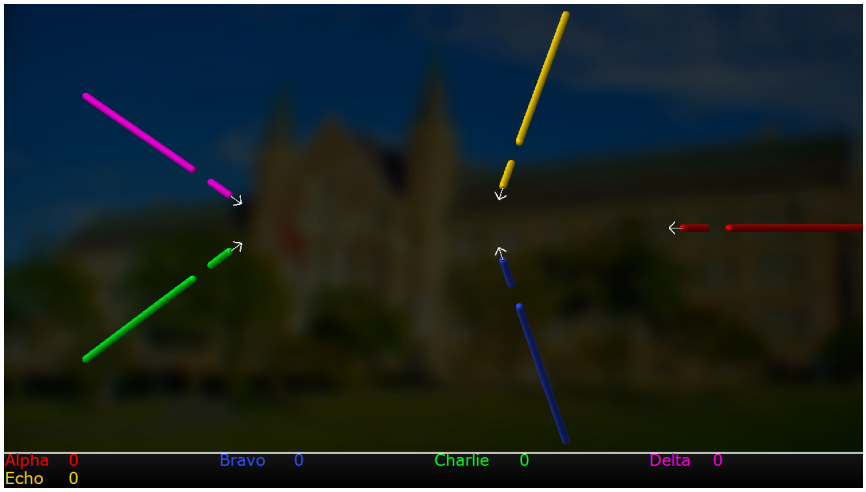


Figure 14.4: *Achtung: game state. The game has just begun and the players are moving in the direction indicated by the arrow on the head of the curve.*

Battle

After the RoundBeginning state ends, the players start to move and must avoid hitting any of the other players. Some players may decide to survive by going away from the other players, while others may try to move to others to trap them in so that they cannot avoid crashing. The game is over when only one player remains.

RoundOver

After the game is over it will pause for a short time to allow the players to look at the scores. This can be seen in Figure 14.5. The game can then either begin a new round, or exit and let the players go back to the MOOSES game selection screen.

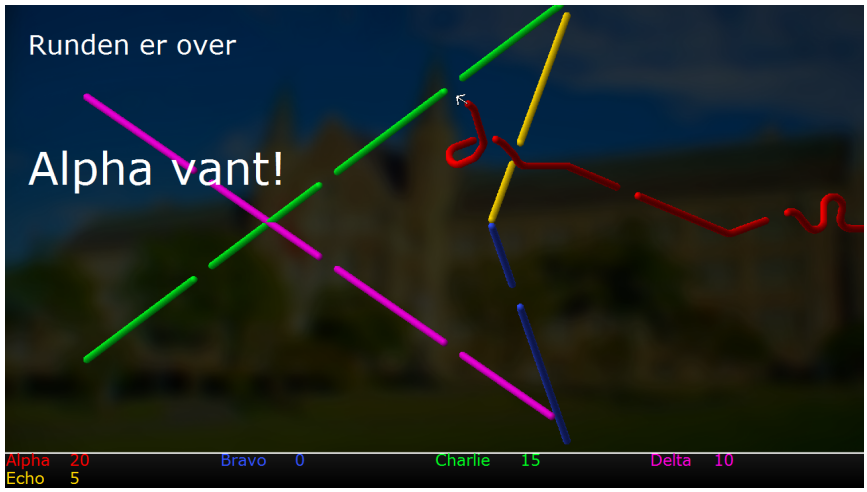


Figure 14.5: *Achtung: round over state. The game is over and "Alpha", the red player, is the winner.*

Part VI

Evaluation and conclusion

CHAPTER

15

EVALUATION

15.1 Playtests

The games were informally tested throughout the course of the semester to ensure find bugs and attempt to balance the games. In addition to this, a group of students from the "NTNU School of Entrepreneurship" were looking for ways to commercialize the MOOSES framework. Two stands were erected at different times to showcase the MOOSES technology, and the three games created in this project were available for testing. The first stand were erected during Kreator'10, an event focusing on innovation and entrepreneurship and the second during the Venture Cup, a business plan competition, final in Mid-Norway. The people who tested the games during these stands were observed and their feedback will be used in addition to the more formal surveys.

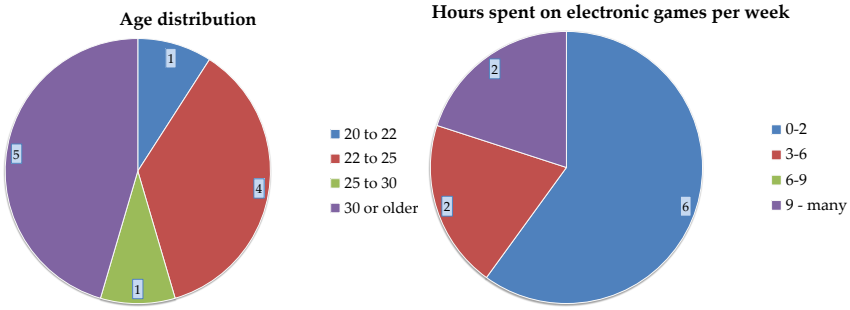
At a later time, two playtests were conducted on different occasions, and the participants filled out a questionnaire. The questionnaire can be found in Appendix E. The entire questionnaire was filled out after the playtests were done. Unfortunately, the playtests were limited in size and thus the results from the questionnaire are not statistically accurate in any way. However, knowing this, and coupled with the observation and oral feedback from the players it is still possible to use the results get a pointer on how the players experienced the different games.

15.1.1 Results of the survey

In this subsection some of the general information gathered from the survey will be presented and commented. The questions regarding a specific game will be presented in that game's section.

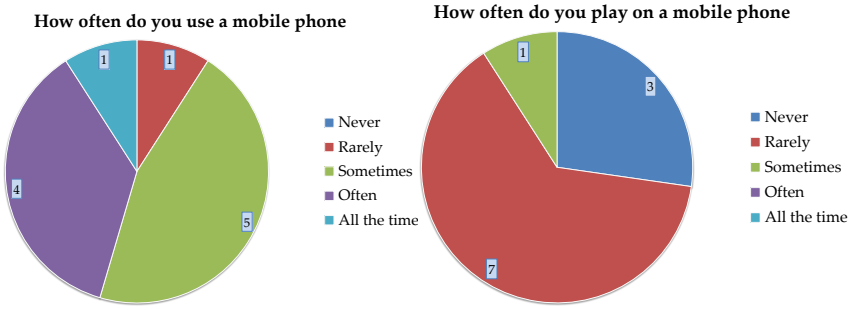
Details about the participants

These questions were created to get some information about the participants of the survey.



(a) Age distribution

(b) Hours spent gaming per week



(c) Mobile phone usage

(d) Playing on the mobile phone

Figure 15.1: *Results regarding the details about the participants.*

Age Survey participants were asked about their age. The distribution can be seen in Figure 15.1a. About half of the participants were the age of typical students, the other half were somewhat older.

Time spent on video games per week To see how familiar the participants were to gaming they were asked how often they played video games. As

seen in Figure 15.1b, most participants did not play a lot of video games.

Mobile phone usage To see how familiar they were with a mobile phone they were asked to estimate their mobile phone usage. Figure 15.1c shows that most of the participants use a mobile phone regularly. They might therefore be familiar with the key layout and general phone usage, which will help when trying to play games with it.

Playing on the mobile phone If the participants were using their mobile phone as a gaming platform it may be easier for them to use MOOSES. However, the participants rarely used their phone to game with, as seen in Figure 15.1d.

General questions about MOOSES concept

In order to find out what the participants thought of the general MOOSES concept a few statements were included in the questionnaire, which they could indicate how they agreed or disagreed with. In this section the statements will be presented along with the results of the survey and comments.

It is easy to use the mobile phone as a gaming controller. While most participants agreed that the mobile phone was suitable as a game controller, see Figure 15.2a, some disagreed or did not form an opinion. In order to improve on this, control schemes using the mobile phone should be well thought out and created for the phone and not just adapted from a regular console controller. s

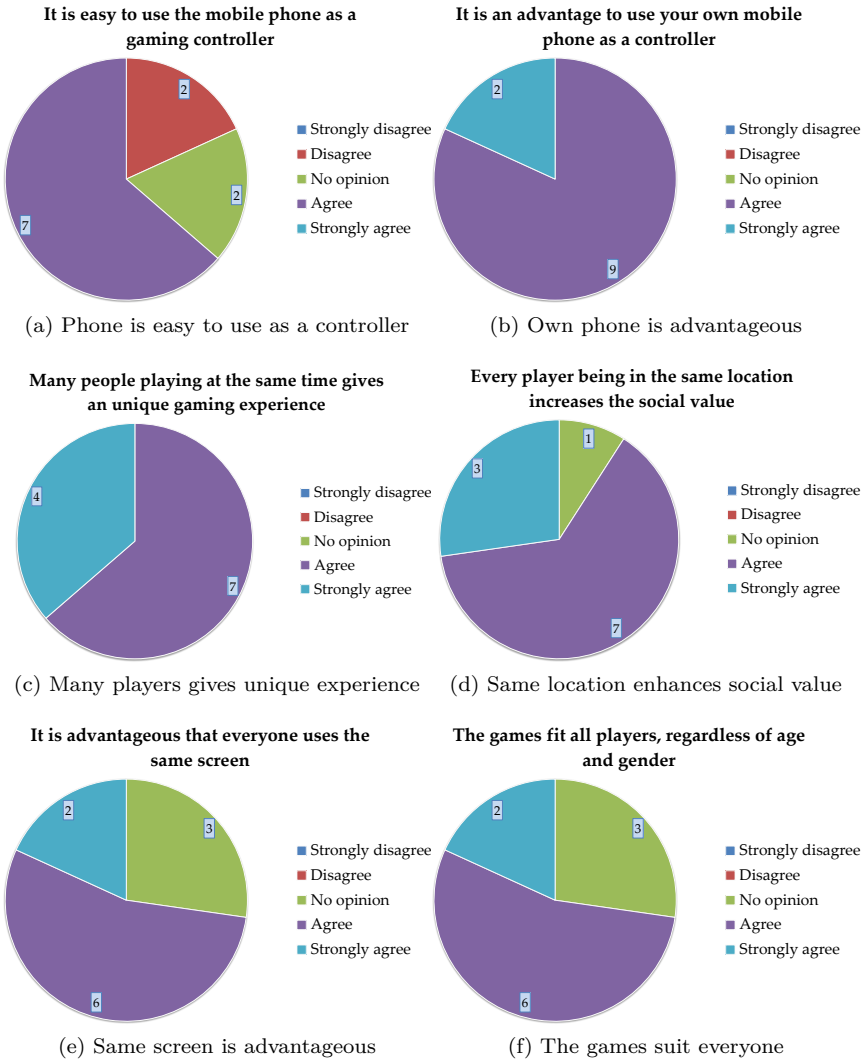
It is an advantage to use your own mobile phone as a controller Everyone agreed that if one were to use a phone as a controller, it is best if you can use your own, see Figure 15.2b.

Many people playing at the same time gives an unique gaming experience The participants agreed to the fact that many people are playing together is a unique and positive, see Figure 15.2c.

Every player being in the same location increases the social value Most participants agreed that being in the same room increases the social value and experience when playing the games, see Figure 15.2d.

It is an advantage that everyone is using the same screen That everyone used the same screen were not as agreed upon as the two previous statements, although most were positive, see Figure 15.2e.

I am willing to pay in order to play like this, given it is possible to win a prize This was a statement which the participants were somewhat ambivalent about. As seen in Figure 15.3a, some agreed and some disagreed. Nothing was said or indicated about the price, and one might speculate that this might have affected the results.



(a) Phone is easy to use as a controller

(b) Own phone is advantageous

(c) Many players gives unique experience

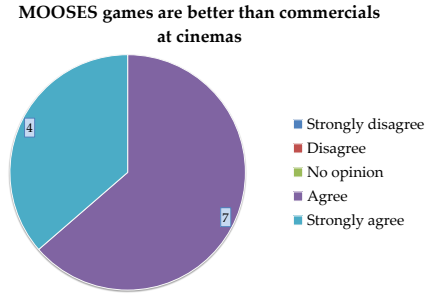
(d) Same location enhances social value

(e) Same screen is advantageous

(f) The games suit everyone

Figure 15.2: *Questions and statements regarding the gaming experience of MOOSES.*

(a) I would pay to play, given a prize



(b) MOOSES games are better than commercials

Figure 15.3: *Questions and statements regarding the commercial potential of MOOSES.*

MOOSES games with product placement is better than watching commercials at a ci
All agreed that playing MOOSES games were better than watching commercials before a movie at a cinema, see Figure 15.3b.

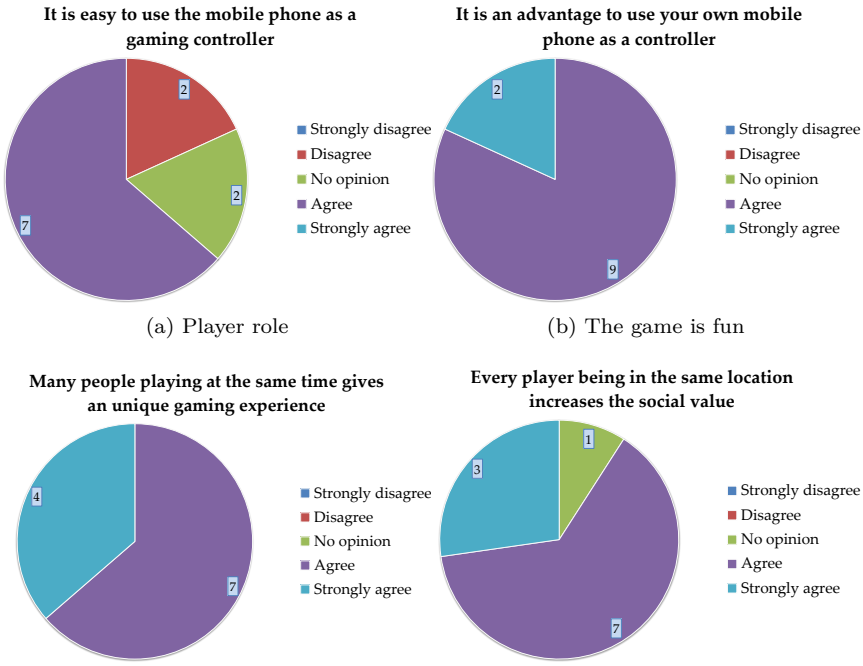
The games suit all players, independent of age and gender Most of the participants agreed that the games they tested could suit everyone, see Figure 15.2f. However, as one participant pointed out, it may be hard for them to say anything about how others will perceive the games, the only thing they can do is guess.

15.2 TowerDefense

TowerDefense was the first game created in this project and is also the most complex game regarding rules, user interface and possible ways to play it. It was supposed to encourage teamwork while each player had control of a personal character. This section will present some positive and negative aspects of the game and suggest some improvements to counter the negative aspects.

Even though the game has some negative aspects, everyone who participated in the survey agreed the game was fun, see Figure 15.4b. Making a game fun can be difficult and it is hard to save a game project with a boring game concept.

The goal of the game was not immediately clear to everyone, as seen in Figure 15.4c. This is a potential issue, as the game should be easy to pick up and learn in short time. To improve on this it may be helpful to have an introductory screen explaining the game mechanics as well as assigning a role to each player before the game begins. As it is now, the game begins and then the player must find their character on the



(a) Player role

(b) The game is fun

(c) Understood the goal quickly

(d) Located my player on the screen easily

Figure 15.4: *Results from statements regarding the general gaming experience with TowerDefense.*

screen and determine what role they have. As seen in Figure 15.4d, this was not easy for everyone either. Letting the players know which role they assume before the game begins should improve on this, as they then know where to look for their character.

Another issue was that seeing how well the team did was not easy, see Figure 15.5a. The players did not see how to win a game, or how far away they or the other team was from winning. Instead of having only numbers indicating respawns and lives, larger progress bars or some other form of visual indication could help.

It may seem as cooperation should been given more emphasis, see Figure 15.5b. Since each player can contribute to the team's success on their own, some of them attempt to do so without coordinating with the others on the team. While the players agreed that cooperation was important for the success of the team (Figure 15.5c), not everyone felt the team could work together (Figure 15.5d). This might be especially true for the attackers, as that role does not really depend on teamwork or cooperation like the defenders. Figure 15.5e shows that players felt that what they did mattered for the success of the team (which of course is important), which might indicate that while working together is not optimal, they made it work by playing on their own in the best interest of the team.

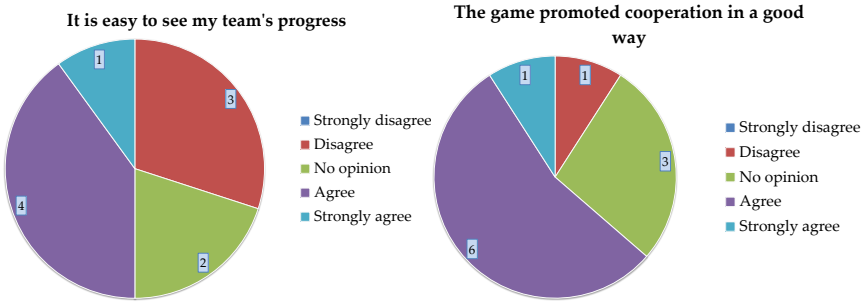
Technically, the game functioned very well in high definition with no visible slowdowns or stuttering. The connection to the MOOSES framework was solid during the playtests and the two stands. There is of course room for improving the graphics as the game does look like a prototype.

15.3 ProductBall

ProductBall was made to test collaborative control of a shared game character, method of cooperation and multiplayer which is fairly uncommon. It is therefore interesting to see if the game was fun to play and if the players thought the method of implementing collaborative play worked.

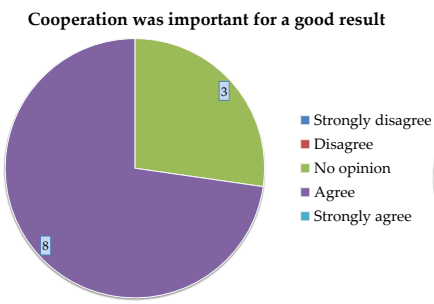
As with TowerDefense, most people found the game to be fun (Figure 15.6a). Players were observed verbally communicating phrases "move faster", "hold up" and "jump", which indicates that game was engaging. Everyone understood the goal easily (Figure 15.6b), which most likely was due to the volleyball concept being familiar.

This game might have been slightly better at promoting cooperation than TowerDefense (Figure 15.6c), probably due to the fact that there is no way around working together. Players thought cooperation were roughly equally important in both games (Figure 15.6d). One could image that cooperation were even more

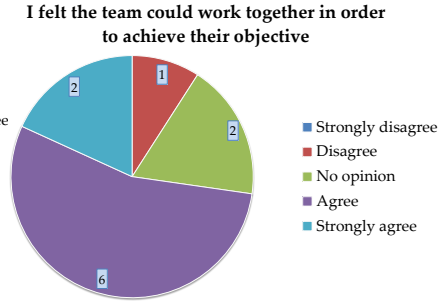


(a) Team progress visible

(b) Cooperation promoted



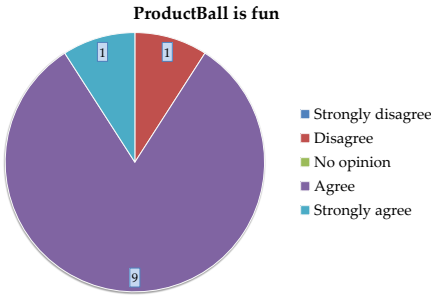
(c) Cooperation important



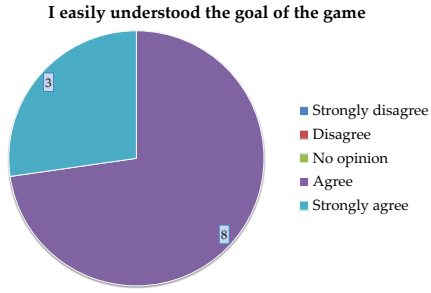
(d) Opportunity to cooperate

(e) Contributed to success

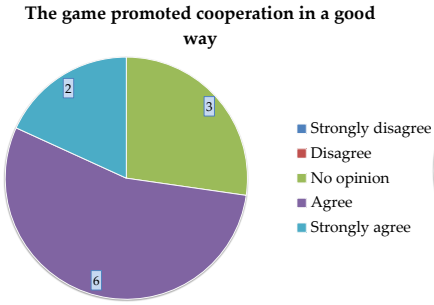
Figure 15.5: *Results from statements regarding the cooperation gaming experience with TowerDefense.*



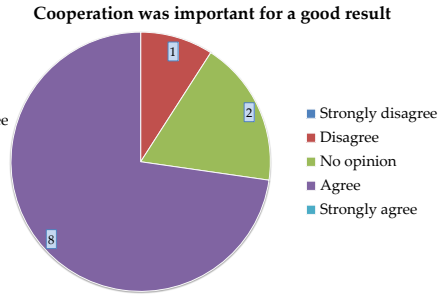
(a) ProductBall is fun



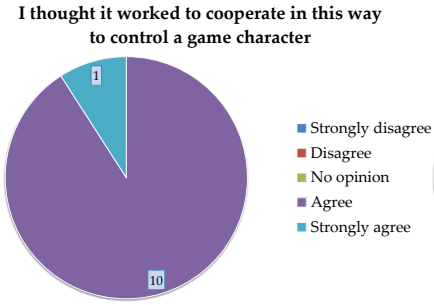
(b) Understood the goal quickly



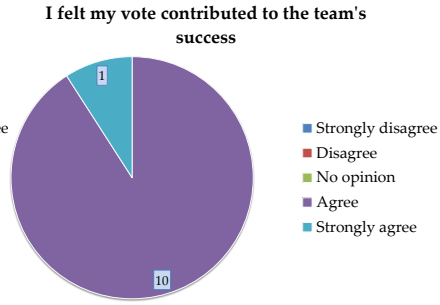
(c) Cooperation promoted



(d) Cooperation important



(e) Collaborative control worked



(f) My voter mattered

Figure 15.6: *Results from statements regarding the gaming experience with ProductBall.*

important in ProductBall, but perhaps the players recognized the fact that as long as most of the team members did the right thing, their actions did not matter too much. Figure 15.6f seems to indicate otherwise however, everyone thought their actions were a contribution to the team. Overall, most of the players recognized that cooperation was important and everyone thought it worked to collaborate on controlling a shared character in this way, see Figure 15.6e.

Implementation of the voting system was also successful. Each player could control an abstract voter and the team character polled each voter attached to them to see what their votes were to take the appropriate action. This scheme worked very well and can be recommended should anyone attempt to create more collaborative games. Giving the characters the possibility to take non-discrete actions such as "slightly left" provided a layer of depth which proved to be successful. The game characters move at a variable speed depending on how many of the team's players are voting in one direction. If all players are voting in the same direction, the character will move at maximum speed.

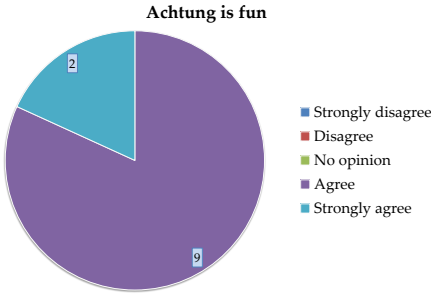
Given a slightly coordinated group of people on both teams, the game would be very easy. Matches could as long as the teams bother to counter the other team. To counter this, a system to increase the difficulty incrementally was implemented. The idea was to make the ball go faster and faster, until one of the teams had to lose either because it became too difficult or impossible to catch the ball. This system was added at a late stage in development and was never functioning perfectly. The version currently in the game makes the ball go only horizontally (by design) faster which does give some weird movements, however if the ball was allowed to go faster vertically as well the ball would go out of sight for longer periods which would be boring. This system should be improved.

15.4 Achtung

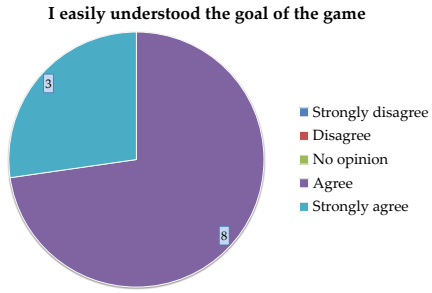
Achtung is based on an old game for DOS called Achtung Die Kurve or Zatacka. The intention with Achtung was to explore a free for all game where each player only was looking out for themselves. Being a remake of an old game, many players were familiar with it and were excited to try the game in a new setting. Since the original already was a shared screen with many players type of game, there was no need to adapt the game concept to MOOSES.

When demonstrating MOOSES on the two different stands, Achtung was the most popular game by far. The familiarity and simple concept made the game a winner. The players who participated in the playtests also found Achtung to be a fun game, see Figure 15.7a. The goal was clear and easy to grasp (Figure 15.7b).

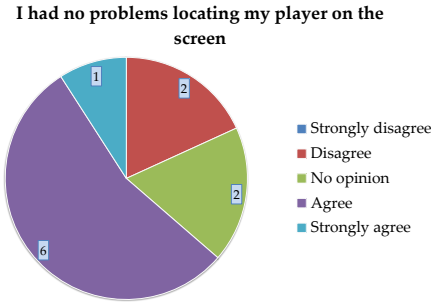
There were two issues with Achtung that should be looked at. The first is a user



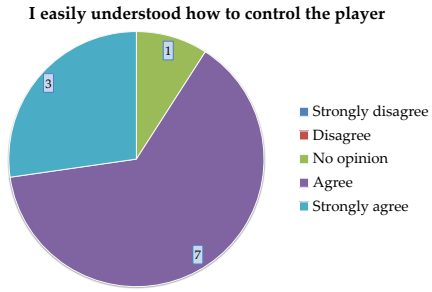
(a) Achtung is fun



(b) Understood the goal quickly



(c) Located my player on the screen easily



(d) Understood the controls easily

Figure 15.7: *Results from statements regarding the gaming experience with Achtung.*

interface issue. Some people had trouble finding their character on the screen, see Figure 15.7c. To find their character, players first had to find their name on the scoreboard, see which color their name was written in, and then look for the character with that color. While not terribly difficult, it can be easily fixed by displaying their name beside the character during the "about to begin" phase. As one of the survey participants pointed out, the current method does not consider color blind persons. He had to find his character by looking for a character which moved according to his actions.

The second issue is that it is too rewarding to play defensively. This is a gameplay issue which does not have an easy fix. Some players suggested increasing the speed; this will make the game harder, but still have the same issues. A better suggestion was to add power ups, which would encourage players to risk playing offensively. These power ups could be scattered around the play field and be picked up by running over them. Adding power ups would add a new layer of depth and tactical play which could be fun.

Achtung was the last game to be developed and really showed how easy it was to implement a game for MOOSES with FMF created in this project. The game was first implemented with keyboard input as the only way to control the characters, and then MOOSES input was added after. More about integrating FMF in a game in Section 15.5.

15.5 Flash for MOOSES Framework

All three games created used the Flash for MOOSES Framework also created in this project. The intent with FMF was to make it easy to create games for MOOSES using Flash. To do this FMF had to be able to give games a stable and consistent interface. FMF had to be specific enough to make things simpler, but also generic enough to not exclude any specific game types.

15.5.1 How to integrate a game with FMF

To illustrate how easy it is to make a Flash game work with MOOSES a step by step recipe on how to do it will be presented.

Before listing what needs to be done it is important to note that FMF assumes that the game must have some internal representation of a player implemented in a class, and that this class also is or can be responsible for handling player input. It could be an Attacker class like in TowerDefense, or a Voter object like in ProductBall. FMF also assumes that all game code can be halted until a connection

to MOOSES has been established. To make this less abstract, the game to give MOOSES capabilities is a hypothetical racing game. The steps presented are still very high level, a more detailed and low-level set of instructions can be found in Appendix B.

1. Make the internal player representation, the Car class, extend and implement the MoosesUnit interface.

2. Initialize FMF in main method and make the game wait until a connection has been made.

3. In the beginning of a round, iterate over the list of connected players, provided by the MoosesData class. For each MoosesPlayer, create a Car object (the internal representation of the player) and it to the MoosesPlayer object. By doing this, input from the players can now be handled by the Car object.

   To support late joining, listen for join messages from the MoosesData, and create a Car object and attach it to the MoosesPlayer object.

4. After a game is over, the game should send score data to MOOSES so that the built-in high score list shows something interesting. Unfortunately, at the moment this is not working correctly, but it is included for completeness.

Using FMF it should be possible to make almost any already functioning game playable through MOOSES in less than an hour.

15.5.2 Further work

The make the MOOSES support complete the testing of sending messages and data from the games to the MOOSES framework should be properly tested. Especially important is sending player scores back to MOOSES. The functions needed have been added to FMF, they are however no functioning perfectly.

15.6 Flash technology

Using Flash and the ActionScript3 language has presented no significant challenges. The development tools used were very similar to Eclipse and Visual Studio, and the compile process is a simple click. Since Flash is more or less designed to be contained in a web browser there were a few quirks to work around. To automatically make the Flash game enter full screen, the game had to be run in the debug player from Adobe. Also, due to some security and MOOSES issues two helper

applications had to be created, see Section 10.3. Once countered, these issues were no longer a concern.

While the performance on Flash is slightly worse compared to technologies like XNA the games created in this project have not had any performance issues, even in full HD resolution. Also, games with better and more advanced graphics have been created in Flash, so with the proper tools and optimization Flash should perform more than well enough for most game types. An issue with Flash is that it does not have any proper 3D rendering capabilities, and the hardware acceleration is limited.

Flash can be an excellent platform to create MOOSES games on. The three games created in this project support this.

CHAPTER

16

RESEARCH QUESTIONS

This chapter will answer the research questions presented in Chapter 3.

RQ 1: What must be done in order to create a game in Flash for the MOOSES framework?

A) How should the MOOSES framework and the game communicate?

By studying previous game implementations in XNA it was found that they used a TCP connection to communicate with the framework. Since TCP connections are technology independent, this is an acceptable way of communicating with the framework. Other methods, if they exist, would be far to time-consuming to explore considering the suitability of the already existing TCP connection.

B) What modifications must be done with the MOOSES framework to allow communication with a Flash game?

Since the Flash games can use the TCP connection developed for XNA games, there is no need to make special modifications to the framework in order to communicate

with Flash. MOOSES cannot launch Flash files directly, but a workaround with an intermediary executable has been created. This works sufficiently well and the MOOSES framework does not need to be modified.

C) What is needed of both experience and software to develop a MOOSES game in Flash?

The scripting language, ActionScript3, is very similar to common programming languages today and should not take much time to get familiar with.

There are two approaches to creating Flash games. The first one is to use the suite of programs developed by Adobe. This is probably the best option, and most widely used among professional developers today, however it is expensive. The second option is to use a free compiler from Adobe called Flex, along with an optional development environment. The Flex compiler along with the IDE called FlashDevelop was used in this project.

RQ 2: What challenges does MOOSES development in Flash pose?

A) How does a Flash game perform in a high definition resolution?

The games have been tested in high definition resolution and were running at satisfactory frame rates. If the frame rates lowered at all it was due to bad code rather than the stress of rendering.

B) What constraints does an implementation in Flash have compared to previous implementations of MOOSES games?

Previous games in Java, C++ and C# are able to use hardware rendering to take a load of the CPU. Flash games cannot do this and is at the mercy of the developers of the Flash platform, Adobe, to ensure good performance. Partly due to this, Flash games cannot render 3D graphics.

Implementations of the Flash platform has been created for the major operating systems, however the performance can be different on each operating system. Especially Linux-based systems suffer from a poorer implementation compared to the Windows version.

RQ 3: How should the modules created in this project be organized for rapid development of new Flash games for the MOOSES framework?

A) What parts of the prototype game should be reusable components for new games?

The direct communication with the MOOSES framework has been separated from the games to a set of modules and helper applications named FMF, see Chapter 10. These modules handle communication, as well as maintain a list of connected players. They also allows the Flash runtime to connect with MOOSES and help the execution of the Flash games. These modules are generic and are intended to be used in new Flash MOOSES games.

B) What directions and recommendations for new projects should be formalized?

A step by step guide for using FMF can be found in Appendix B to aid in creating new Flash projects with MOOSES.

RQ 4: How is the user experience when playing MOOSES games?

A) Is shared control over a character fun and enjoyable?

Results from the testing of ProductBall seem to indicate that collaborative control of a character not only works well, but also is fun. More about the experience with ProductBall in Section 15.3.

B) How do players experience the different multiplayer modes?

The feedback from the survey suggests that all three games and their multiplayer modes were fun. When observing the players playing the different games there were a few differences. Players communicated verbally with each other while playing all games, but the communication were different when playing ProductBall compared to TowerDefense and Achtung. The communication was almost exclusively within the team, trying to coordinate the control of the character. In TowerDefense and Achtung, players would rather comment on the skill of the opponent or exclaim

words of frustration. There were few teamwork messages when playing TowerDefense unfortunately. This might have been due to the apparent lack of need to coordinate efforts to win. The three multiplayer modes were all fun, the difference lies in who players are communication to and the nature of the communication.

C) Does cooperation work in a MOOSES game?

Nothing with MOOSES prevents cooperative games. The trick is to design a game which is both and encourages teamwork. Both ProductBall and TowerDefense are games where cooperative gameplay works well, but they are rather simple and could benefit from polish to really shine. Cooperative gameplay, especially collaborative, should be explored further in the future.

CHAPTER

17

CONCLUSION

This project had three connected parts: develop components to facilitate Flash games for MOOSES, test these components by either creating new games or modifying existing games, and finally evaluate the games created and the Flash platform.

A set of classes and helper applications called Flash for MOOSES Framework were created. The purpose of these classes was to make it easy to produce MOOSES games in Flash. These classes were designed so that they would be easy and intuitive to integrate and still generic enough to not set any restrictions on the type of games. FMF could potentially have covered more areas such as the game loop, drawing and sound, but these areas are not difficult in Flash development, even for beginners, and they are much harder to get just right in a framework.

FMF was tested by creating three games, each with a different multiplayer mode. These games were tested through playtests and feedback was collected through a survey. The games received a positive response and the overall opinion was that they were all fun to play. The different multiplayer modes make people communicate in different patterns and with different purposes. The verbal communication when playing Achtung was either to comment other players or to exclaim frustration. During ProductBall people would communicate within the team trying to coordinate the control of the character, and while playing TowerDefense communication resembles the one in Achtung although there was some attempt of coordination.

The current implementation of FMF makes communication with MOOSES and integrating a existing or new game very easy. Modifying a completed game to be MOOSES compatible should take less than one hour unless there are unforeseen issues.

The Flash platform work very well for MOOSES. It is easy to write games due to the simplicity and familiarity of ActionScript3 and the zero-to-game factor is very good. There are a lot of resources available to aid when creating games, as well as message boards and forums. The only potential issue is the performance of Flash and the lack of 3D rendering support.

CHAPTER

18

FURTHER WORK

The goal of each game was understood quickly, but it had to be told by the ones hosting the stand or the playtests. It could be a good idea to have a quick walk-through of the game goals and controls before the game starts. Hopefully this would clear up confusion regarding some of the game mechanics.

The three games developed must implement a good way to deal with infinite matches and defensive gameplay. In ProductBall's case, the game could last as long as the players wanted to, unless some method of increasing the difficulty over time was implemented. The current implementation is slightly flawed in the sense that the ball moves oddly after a while, but does its purpose. This method should be improved upon.

Regarding Achtung, some players would play very defensively, and creating a safe pattern on "their" side of the game screen. This is an effective, but for everyone else boring, tactic. To counter this, one could spawn power ups with different effects randomly on the playing field. These power ups may either be so beneficial to have that everyone would want to get them, or they might make defensive gameplay useless.

TowerDefense could use a bit of polishing in both graphics and gameplay department if it were to be used further. The graphics need a clearer direction and better artwork, but as a prototype they are sufficient. The gameplay could use incentives to avoid the well-travelled path.

Collaborative games with shared control should be tested further. This form of gameplay is very rare, and could become a unique aspect of MOOSES. Collaborative control worked well in ProductBall, and the same scheme could be applied to many different games. Another scheme could be individual control of a part of greater entity. As an example, one could have a canoe race. There could be one canoe per team, and each team could consist of two to four rowers controlled by a player. Each player could then choose to row on either side of the canoe, but unless their efforts are coordinated they might lose speed or just steer off course.

The Flash for MOOSES Framework should have a proper test of the talk back functions. As of now the FMF can send messages to MOOSES, but the implementation has not been tested on a larger scale.

Finally, it would be beneficial to create a generic and game independent phone client. Although the phone client can be used as a private screen, there are very few games where this is needed. Furthermore it forces the users to shift focus back and forth between the phone and the screen which may cause the to lose track of their player and causing unnecessary frustration.

Part VII

Appendices

APPENDIX

A

TERMS AND ABBREVIATIONS

2D 2-dimensional

3D 3-dimensional

AI Artificial Intelligence

CD Compact Disk

CPU Central Processing Unit

FMF Flash for MOOSES Framework

FPS First Person Shooter

GB Gigabyte

GHz Gigahertz

HD High definition

HDD Harddisk drive

HTML HyperText Markup Language

IDE Integrated Development Environment

IDI Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU)

IP Internet Protocol

J2ME Java 2 Platform, Micro Edition

LAN Local Area Network

MB Megabyte

Mbps Megabits per second

MHz Megahertz

MMOG Massively Multiplayer Online Game

MMOFPS Massively Multiplayer Online First-Person Shooters

MMORPG Massively Multiplayer Online Role-Playing Game

MMORTS Massively Multiplayer Online Real- Time Strategy Games

MP3 MPEG-1 Audio Layer 3

MOOSES Multiplayer On One Screen Entertainment System

NTNU Norwegian University of Science and Technology

PC Personal Computer

RAM Random-access memory

RPG Role-Playing Game

RTS Real-Time Strategy

RQ Research Question

TCP Transmission Control Protocol

TV Television

USB Universal Serial Bus

UML Unified Modeling Language

WAV Waveform audio format

WMA Windows Media Audio

XML Extensible Markup Language

APPENDIX

B

HOW TO INTEGRATE FMF WITH YOUR GAME

This section will describe how to integrate the FMF with a game written in ActionScript3. It assumes you have the skills to create a simple game using the development approach of you choice. Also, it is assumed that you have added the code files to your project.

Add the following code statements to the top of your *Main* class (or any class which persists through all stages of the game, in this how-to that class will be called Main), before the class description.

```
import no.ekse.mooses.MoosesConnectionEvent;
import no.ekse.mooses.MoosesData;
import no.ekse.mooses.MoosesFrameworkClient;
import no.ekse.mooses.MoosesInfo;
```

This will give your class access to the necessary FMF classes. Add the following statements to your class. It is important to add them to the class and not a function due to scoping.

```
private var mooses:MoosesFrameworkClient;
```

```
private var moosesLog:MoosesInfo;
private var moosesData:MoosesData;
```

In your *Main* function, usually placed in your *Main* class, add the following statements to initialise the FMF modules.

```
mooses = new MoosesFrameworkClient();
moosesLog = new MoosesInfo(mooses);
moosesData = new MoosesData(mooses);

mooses.addEventListener(MoosesConnectionEvent.MOOSES_CONNECTED,
            MoosesHandleConnected);
mooses.addEventListener(MoosesConnectionEvent.MOOSES_DISCONNECTED,
            MoosesHandleDisconnected);
```

Depending on the user interface you have created for you game, you might not want to connect to the MOOSES game server right away. For games solely for MOOSES however, the first thing you want to do is connect. This can be done by adding *mooses.connect();* after *mooses* has been constructed. *mooses.connect();* attempts to open a TCP connection to the default MOOSES game server, which is locally on the port 1900. For external game servers, use *mooses.connectTo(serverAddress, serverPort);*.

The two last statements in the code block above says that the function *MoosesHandleConnected* will be executed when mooses triggers the event *MoosesConnectionEvent.MOOSES_CONNECTED*, and that the function *MoosesHandleDisconnected* will be executed when mooses triggers the event *MoosesConnectionEvent.MOOSES_DISCONNECTED*.

The actual game mechanics should not start before *MoosesHandleConnected* is executed. When executed, you know that your connection with MOOSES is fully operational. This is all there is to do in the *Main* class.

It is a good idea to not start the game right away after receiving a connection, as more players might want to join in. Therefore, the game should begin in pause mode, and only start after a set time.

At this point it becomes harder to explicitly say what you should do. However, each game unit which a player is to control must implement the interface called *MoosesUnit*. In the beginning of each round, a you can attach an unit implementing *MoosesUnit* to a *MoosesPlayer* by calling the *AttachUnit(unit)* function to allow that player to control the unit. An unit can later be detached from a player by using *DetachUnit()*. You can get the list of players connected by using *moosesData.GetPlayerList();*, which returns an array of *MoosesPlayer*.

Now you get input from the MOOSES controller directly in you game unit. Following is a block of code which gives an example of how movement can occur.

```
import no.ekse.mooses.MoosesPlayer;
import no.ekse.mooses.MoosesUnit;
import no.ekse.mooses.MoosesFrameworkClient;

public class GameUnit extends Sprite implements MoosesUnit
{
  ...
  public function KeyDown(keyId:int):void {
    if (keyId == MoosesFrameworkClient.KEY_4) {
      x -= 64;
    } else if (keyId == MoosesFrameworkClient.KEY_6) {
      x += 64;
    } else if (keyId == MoosesFrameworkClient.KEY_2) {
      y -= 64;
    } else if (keyId == MoosesFrameworkClient.KEY_8) {
      y += 64;
    }
  }
  ...
}
```

Every time a user clicks the 4 button on his or hers mobile phone, the GameUnit is moved 64 pixels to the left.

The game should now connect to the MOOSES game server, as well as respond to input from the players. From this point on, it is up to you.

APPENDIX

C

DATA ATTACHEMENT

Source Code/ Folder containing all the source code developed in this project.

Debug Executables/ Folder containing debug versions of the three games. These can be opened in a web browser and watched. Will most likely do random things, but are there for completeness.

APPENDIX

D

SOURCE CODE

Included in the sections below is the source code for the Flash For MOOSES Framework. Source code for the games can be found in the attached data.

MoosesFrameworkClient.as

```
ï»¿package  no.ekse.mooses
{
  import flash.events.EventDispatcher;
  import flash.net.Socket;
  import flash.utils.ByteArray;
  import flash.events.Event;
  import flash.events.IOErrorEvent;
  import flash.events.ProgressEvent;


  /**
   * The main module which maintains a connection between Flash and the MOOSES
       framework.
   * @author Magnus Ekse
   */
  public class MoosesFrameworkClient extends EventDispatcher
  {

    public static const KEY_STATE_UP:int = 1;
    public static const KEY_STATE_DOWN:int = 0;
    public static const KEY_STATE_SPECIAL:int = 2;

    public static const KEY_0:int = 0;
    public static const KEY_1:int = 1;
```

```
public static const KEY_2:int = 2;
public static const KEY_3:int = 3;
public static const KEY_4:int = 4;
public static const KEY_5:int = 5;
public static const KEY_6:int = 6;
public static const KEY_7:int = 7;
public static const KEY_8:int = 8;
public static const KEY_9:int = 9;
public static const KEY_STAR:int = 10;
public static const KEY_HASH:int = 11;
public static const KEY_A:int = 12;
public static const KEY_B:int = 13;

private var sck:Socket = new Socket();

private var remoteAddress:String;
private var remotePort:int;

private var hasReceivedStartupSignal:Boolean = false;


public function MoosesFrameworkClient() {

  trace("Mooses: Initializing framework client");

  remoteAddress = "localhost";
  remotePort = 1900;


  trace("Mooses: Framework client ready");

}

public function initSocket(a:String, p:int):void {

  trace("Mooses: Connecting to socket: " + a + ":" + p);
  dispatchEvent(new MoosesConnectionEvent(MoosesConnectionEvent.
      MOOSES_ATTEMPTCONNECT, false, false));

  sck = new Socket(a, p);

  sck.addEventListener(Event.CONNECT, socketConnect);
  sck.addEventListener(Event.CLOSE, socketClose);
  sck.addEventListener(IOErrorEvent.IO_ERROR, socketIOError);
  sck.addEventListener(ProgressEvent.SOCKET_DATA, socketReceive);

}


public function connectedToFramework():Boolean {

  trace("Mooses: Checking if connected to framework");

  if (hasReceivedStartupSignal) {
    trace("Mooses: Has received startup signal");

    if (sck.connected) {
      trace("Mooses: Socket is connected, connection OK");

      return true;
    }
  }

  trace("Mooses: Not connected to framework");

  return false;

}
```

```
public function connect():void {

  trace("Mooses: Attempting to connect to " + remoteAddress + ":" +
       remotePort);

  hasReceivedStartupSignal = false;
  initSocket(remoteAddress, remotePort);

}


public function connectTo(address:String, port:int):void {

  trace("Mooses: Attempting to connect to " + address + ":" + port);

  hasReceivedStartupSignal = false;
  initSocket(address, port);

}


private function socketConnect(e:Event):void {

  trace("Mooses: Socket connected");

}


private function socketClose(e:Event):void {

  trace("Mooses: Socket closed");
    dispatchEvent(new MoosesConnectionEvent(MoosesConnectionEvent.
         MOOSES_DISCONNECTED, false, false));
  if (hasReceivedStartupSignal) {
  }
}

/*
 * Will buzz the controller of the specified player for a
 * duration specified in miliseconds
 *
 * How the buzz is implemented is up to the controller, a mobile
 * phone will vibrate.
 */
public function buzzPlayer(playerId:int, buzzDuration:int):void {

  if (hasReceivedStartupSignal && sck.connected && playerId < 9000) {

    sck.writeInt(1020);
    sck.writeInt(playerId);
    sck.writeInt(buzzDuration);

    sck.flush();
  }

}

/*
 * Update the amount of points the player has
 *
 */
public function updatePlayerScore(playerId:int, score:int):void {

  if (hasReceivedStartupSignal && sck.connected && playerId < 9000) {

    trace("Mooses: Send: Type: " + 1030);
```

```
    sck.writeInt(1030);

    trace("Mooses: Send: Player: " + playerId);
    sck.writeInt(playerId);

    trace("Mooses: Send: Score: " + score);
    sck.writeInt(playerId);

  }
}

/*
 * Sends a message to the player. How this message is interpreted is up to
     the client script.
 * This function is therefore as general as possible to allow maximum
     usability. Shorter and
 * more concrete functions to update the client should be made on a per-game
      basis.
 *
 * Only two types of data is supported; integer and string.
 */
public function sendMessageToPlayer(playerId:int, messageType:int,
     messageData:Array):void {

  if (hasReceivedStartupSignal && sck.connected && playerId < 9000) {

    trace("Mooses: Send: Type: " + messageType);
    sck.writeInt(messageType);

    trace("Mooses: Send: Player: " + playerId);
    sck.writeInt(playerId);

    //sending the number of content units, not necessarily the size
    trace("Mooses: Send: NumPax: " + messageData.length);
    sck.writeByte(messageData.length);

    for (var i:int = 0; i < messageData.length; i++) {

      if (messageData[i] is String) {
        //this part is guesswork
        trace("Mooses: Send: Pax " + i + ": String: " + (messageData[i] as
            String));
        sck.writeByte(5); // 5 -> next data is string
        sck.writeUTF(messageData[i] as String);
        sck.writeByte(0);

      } else if (messageData[i] is int) {
        trace("Mooses: Send: Pax " + i + ": Int: " + (messageData[i] as int)
            );
        sck.writeByte(2); // 2 -> next data is int
        sck.writeInt(messageData[i] as int);

      }

    }

    sck.flush();

    trace("Mooses: Send: Done");

  }

}

private function flushRemainingBytes():void {

  var b:ByteArray = new ByteArray();
  sck.readBytes(b, 0, sck.bytesAvailable);
```

```
    trace(b);

  }

  private function socketReceive(e:ProgressEvent):void {

    trace("Mooses: Socket received data, bytes in buffer = " + sck.
        bytesAvailable);

    var bytesActuallyRead:int = 0;

    if (sck.bytesAvailable < 8) {
      trace("Mooses: Too little data in the incoming buffer, skipping this and
          waiting for more");

      return;
    }

    if (!hasReceivedStartupSignal) {

      //Expected data in the buffer is two int values.
      trace("Mooses: Startup signal recieved, " + sck.readInt() + " . " + sck.
          readInt());

      //Should be good to go now.
      hasReceivedStartupSignal = true;
      dispatchEvent(new MoosesConnectionEvent(MoosesConnectionEvent.
          MOOSES_CONNECTED, false, false));

    }
    if (hasReceivedStartupSignal) {

      while (sck.bytesAvailable >= 12) {

        trace("bytes available " + sck.bytesAvailable);

        if (sck.bytesAvailable < 4) { flushRemainingBytes(); return; }
        var packetId:int = sck.readInt();
        trace("packetId " + packetId);

        if (sck.bytesAvailable < 4) { flushRemainingBytes(); return; }
        var playerId:int = sck.readInt();
        trace("playerId " + playerId);
        //Get the size of the packet (without the header).

        if (sck.bytesAvailable < 4) { flushRemainingBytes(); return; }
        var packetSize:int = sck.readInt();
        trace("packetSize " + packetSize);

        bytesActuallyRead = sck.bytesAvailable;

        trace("Mooses: Packet received: " + packetId + ", " + playerId + ", "
            + packetSize);

        if (packetId == 1000) {
          //Server started

        } else if (packetId == 1010) {
          //Player joins
          var playerName:String = sck.readUTF();

          trace("Mooses: Player joined, id = " + playerId + ", name = " +
              playerName);

          trace(sck.readByte());

          dispatchEvent(new MoosesPlayerEvent(MoosesPlayerEvent.
              MOOSES_PLAYER_JOIN, playerId, playerName, false, false));
```

```
} else if (packetId == 1011) {
  //Player quits.

  trace("Mooses: Player quits, id = " + playerId);

  dispatchEvent(new MoosesPlayerEvent(MoosesPlayerEvent.
      MOOSES_PLAYER_QUIT, playerId, "", false, false));

} else if (packetId == 40) {
  //Special button. TBD.
  trace("Mooses: Special button pressed, 40");

  dispatchEvent(new MoosesKeyEvent(MoosesKeyEvent.MOOSES_KEY_EVENT,
      playerId, 10, 2, false, false));

} else if (packetId == 80) {
  //Special button. TBD.
  trace("Mooses: Special button pressed, 80");

  dispatchEvent(new MoosesKeyEvent(MoosesKeyEvent.MOOSES_KEY_EVENT,
      playerId, 11, 2, false, false));

} else if (packetId == 0) {
  //Movement key pressed

  if (sck.bytesAvailable >= 5) {

    if (sck.bytesAvailable < 4) { flushRemainingBytes(); return; }

    var keyId:int = sck.readInt();

    if (sck.bytesAvailable < 1) { flushRemainingBytes(); return; }

    var keyState:Boolean = sck.readBoolean();

    trace("Mooses: Movement key packet, key = " + keyId + ", status =
        " + keyState);

    /*  a   b |  12    13
     *  1 2 3 |  01 02 03
     *  4 5 6 |  04 05 06
     *  7 8 9 |  07 08 09
     *  * 0 # |  10 00 11
     */

    //The key ids sent by MOOSES are not particulary logical and I
        suspect them to be
    //dependent on the client. This section _might_ have to be edited
        to conform to
    //different clients for different games.
    if (keyId == 100)
    {
      keyId = KEY_4;
    } else if (keyId == 102)
    {
      keyId = KEY_2;
    } else if (keyId == 101)
    {
      keyId = KEY_6;
    } else if (keyId == 103)
    {
      keyId = KEY_8;
    }

    dispatchEvent(new MoosesKeyEvent(MoosesKeyEvent.MOOSES_KEY_EVENT,
        playerId, keyId, keyState == 0 ? 1 : 0, false, false));
  } else {
```

```
            trace("Mooses Warning: Hmm, erroneous number of bytes left in
                buffer: " + sck.bytesAvailable);
            trace("Mooses Warning: Peeking at mystery byte: " + sck.readByte()
                );
            flushRemainingBytes();
          }
        } else {

          flushRemainingBytes();
        }

        bytesActuallyRead = bytesActuallyRead - sck.bytesAvailable;
        trace("Bytes processed this run:" + (bytesActuallyRead));
      }

    }

    trace("Mooses: Bytes in buffer after handling: " + sck.bytesAvailable);

    flushRemainingBytes();

  }


  private function socketIOError(e:IOErrorEvent):void {

    trace("Mooses: Socket IO error");
    dispatchEvent(new MoosesConnectionEvent(MoosesConnectionEvent.
        MOOSES_ERRORCONNECT, false, false));
  }
 }

}
```

MoosesData.as

```
ï»¿package  no.ekse.mooses
{
  import flash.events.EventDispatcher;
  import no.ekse.mooses.MoosesFrameworkClient;

  /**
   * This module maintains a list of players currently connected to the MOOSES
       framework.
   * When a key event is found, this module sends the key event to the proper
       MoosesUnit.
   *
   * @author Magnus Ekse
   */
  public class MoosesData extends EventDispatcher
  {
    private var client:MoosesFrameworkClient;

    private var players:Array = new Array();

    public function MoosesData(client:MoosesFrameworkClient)
    {

      this.client = client;

      client.addEventListener(MoosesPlayerEvent.MOOSES_PLAYER_JOIN, playerJoin);
      client.addEventListener(MoosesPlayerEvent.MOOSES_PLAYER_QUIT, playerQuit);

      client.addEventListener(MoosesKeyEvent.MOOSES_KEY_EVENT, keyEvent);
```

```
}

public function getPlayerList():Array {

  return players;

}

public function buzzUnit(unitToBuzz:MoosesUnit, duration:int):void {

  for (var i:int; i < players.length; i++) {
    if (players[i].getAttachedUnit() == unitToBuzz) {
      client.buzzPlayer(players[i].getPlayerId(), duration);
    }
  }

}

public function addPlayer(id:int, name:String):void {

  players.push(new MoosesPlayer(this, id, name));

}

public function updatePlayerScore(playerId:int, score:int):void {

  client.updatePlayerScore(playerId, score);
}
public function sendMessageToPlayer(playerId:int, messageType:int,
     messageData:Array):void {

  client.sendMessageToPlayer(playerId, messageType, messageData);

}

public function broadcastMessage(messageType:int, messageData:Array):void {

  for (var i:int; i < players.length; i++) {
    client.sendMessageToPlayer( (players[i] as MoosesPlayer).getPlayerId(),
        messageType, messageData);
  }

}

public function removePlayer(id:int):void {

  var playerId:int = id;
  var playerToRemoveIndex:int;

  for (var i:int; i < players.length; i++) {
    if (players[i].getPlayerId() == playerId) {
      playerToRemoveIndex = i;
    }
  }

  players.splice(playerToRemoveIndex, 1);

}

public function playerJoin(e:MoosesPlayerEvent):void {

  addPlayer(e.playerId, e.playerName);

}
public function playerQuit(e:MoosesPlayerEvent):void {

  removePlayer(e.playerId);
```

```
    }

    public function keyEvent(e:MoosesKeyEvent):void {

      for each (var player:MoosesPlayer in players) {

        if (player.getPlayerId() == e.playerId) {


          if (e.keyState == MoosesFrameworkClient.KEY_STATE_DOWN) {
            //State = 0, pressed.
            player.keyDown(e.keyCode);
          } else if (e.keyState == MoosesFrameworkClient.KEY_STATE_UP) {
            //State = 1, released.
            player.keyUp(e.keyCode);
          } else if (e.keyState == MoosesFrameworkClient.KEY_STATE_SPECIAL) {
            //State = 2, pressed and will not be released.
            player.keyDown(e.keyCode);
          }


        }
      }

    }

  }

}
```

MoosesInfo.as

```
ï»¿package  no.ekse.mooses
{

  import flash.display.DisplayObject;
  import flash.display.MovieClip;
  import flash.display.Sprite;
  import flash.text.TextField;
  import flash.text.TextFormat;
  import flash.utils.getTimer;

  public class MoosesInfo extends Sprite
  {
    private var textLog:TextField = new TextField();
    private var client:MoosesFrameworkClient;

    public function MoosesInfo(client:MoosesFrameworkClient) {
      this.client = client;
      textLog.autoSize = "left";

      log("Flash for MOOSES\n");

      client.addEventListener(MoosesConnectionEvent.MOOSES_ATTEMPTCONNECT,
          moosesAttemptConnect);
      client.addEventListener(MoosesConnectionEvent.MOOSES_ERRORCONNECT,
          moosesConnectionError);
      client.addEventListener(MoosesConnectionEvent.MOOSES_CONNECTED,
          moosesConnected);
      client.addEventListener(MoosesConnectionEvent.MOOSES_DISCONNECTED,
          moosesDisconnected);
      client.addEventListener(MoosesPlayerEvent.MOOSES_PLAYER_JOIN,
          moosesPlayerJoin);
```

```
      client.addEventListener(MoosesPlayerEvent.MOOSES_PLAYER_QUIT,
          moosesPlayerQuit);
      client.addEventListener(MoosesKeyEvent.MOOSES_KEY_EVENT, moosesKey);

      addChild(textLog);
    }

    public function log(s:String):void {
      textLog.appendText("\n" + s);
      trace("MoosesLog: " + s);
    }
    public function moosesKey(e:MoosesKeyEvent):void {
      log("Key event: key [" + e.keyCode + "] state [" + e.keyState + "] player
          id [" + e.playerId + "]");
    }
    public function moosesPlayerJoin(e:MoosesPlayerEvent):void {
      log("Player connected: " + e.playerName + ":" + e.playerId);
    }
    public function moosesPlayerQuit(e:MoosesPlayerEvent):void {
      log("Player disconnected: " + e.playerName + ":" + e.playerId);
    }
    public function moosesAttemptConnect(e:MoosesConnectionEvent):void {
      log("Attempting to connect to framework");
    }
    public function moosesConnected(e:MoosesConnectionEvent):void {
      log("Connected to framework");
    }
    public function moosesDisconnected(e:MoosesConnectionEvent):void {
      log("Disconnected from framework");
    }
    public function moosesConnectionError(e:MoosesConnectionEvent):void {
      log("Connection error");
    }

  }
}
```

MoosesConnectionEvent.as

```
ï»¿package  no.ekse.mooses
{
  import flash.events.Event;

  /**
   * The event which is propagated when the connection between Flash and MOOSES
       changes.
   *
   * @author Magnus Ekse
   */
  public class MoosesConnectionEvent extends Event
  {
    public static const MOOSES_CONNECTED:String = "MoosesConnected";
    public static const MOOSES_DISCONNECTED:String = "MoosesDisconnected";
    public static const MOOSES_ATTEMPTCONNECT:String = "MoosesAttemptConnect";
    public static const MOOSES_ERRORCONNECT:String = "MoosesErrorConnect";

    public function MoosesConnectionEvent(type:String, bubbles:Boolean=false,
        cancelable:Boolean=false)
    {
      super(type, bubbles, cancelable);

    }

    public override function clone():Event
    {
```

```
      return new MoosesConnectionEvent(type, bubbles, cancelable);
    }

    public override function toString():String
    {
      return formatToString("MoosesConnectionEvent", "type", "bubbles", "
          cancelable", "eventPhase");
    }

  }

}
```

MoosesKeyEvent.as

```
ï»¿package  no.ekse.mooses
{
  import flash.events.Event;

  /**
   * ...
   * @author Magnus Ekse
   */
  public class MoosesKeyEvent extends Event
  {
    public static const MOOSES_KEY_EVENT:String = "MoosesKeyEvent";

    public var keyCode:int = -1;
    public var keyState:int = -1;
    public var playerId:int = -1;

    public function MoosesKeyEvent(type:String, playerId:int, keyCode:int,
        keyState:int, bubbles:Boolean=false, cancelable:Boolean=false)
    {
      super(type, bubbles, cancelable);

      this.playerId = playerId;
      this.keyCode = keyCode;
      this.keyState = keyState;
    }

    public override function clone():Event
    {
      return new MoosesKeyEvent(type, playerId, keyCode, keyState, bubbles,
          cancelable);
    }

    public override function toString():String
    {
      return formatToString("MoosesKeyEvent", "type", "playerId", "keyCode", "
          keyState", "bubbles", "cancelable", "eventPhase");
    }

  }

}
```

MoosesPlayer.as

```
ï»¿package  no.ekse.mooses
{
```

```
public class MoosesPlayer
{

  //an array holding the current state of all keys
  private var keyState:Array = new Array(16);

  private var playerId:int = new int();
  private var playerName:String = new String("Unnamed");

  private var attachedUnit:MoosesUnit = null;

  private var moosesData:MoosesData;

  public function MoosesPlayer(moosesData:MoosesData, playerId:int, playerName
      :String)
  {
    this.playerId = playerId;
    this.playerName = playerName;
    this.moosesData = moosesData;

    for (var i:int; i < 16; i++) {
      keyState[i] = false;
    }

  }

  public function attachUnit(unit:MoosesUnit):void {
    attachedUnit = unit;
    attachedUnit.setMoosesPlayer(this);
  }
  public function detachUnit():void {
    if (attachedUnit != null)
    {
      attachedUnit.setMoosesPlayer(null);
      attachedUnit = null;
    }
  }
  public function getAttachedUnit():MoosesUnit {
    return attachedUnit;
  }

  /*  a   b |  12    13
   *  1 2 3 |  01 02 03
   *  4 5 6 |  04 05 06
   *  7 8 9 |  07 08 09
   *  * 0 # |  10 00 11
   */
  public function keyDown(keyId:int):void
  {
    keyState[keyId] = true;

    if (attachedUnit != null) attachedUnit.keyDown(keyId);
  }

  public function keyUp(keyId:int):void
  {
    keyState[keyId] = false;

    if (attachedUnit != null) attachedUnit.keyUp(keyId);
  }

  public function getKeyState(keyId:int):Boolean {
    return keyState[keyId];
  }

  public function getPlayerId():int {
    return playerId;
```

```
    }
    public function getPlayerName():String {
      return playerName;
    }

    public function sendScore(score:int):void {

      moosesData.updatePlayerScore(playerId, score);

    }

    public function update():void
    {
      trace("player update");
    }
  }

}
```

MoosesPlayerEvent.as

```
ï»¿package  no.ekse.mooses
{
  import flash.events.Event;

  /**
   * The event which is propagated if a player joins or quits the MOOSES server.
   *
   * @author Magnus Ekse
   */
  public class MoosesPlayerEvent extends Event
  {
    public static const MOOSES_PLAYER_JOIN:String = "MoosesPlayerJoin";
    public static const MOOSES_PLAYER_QUIT:String = "MoosesPlayerQuit";

    public var playerId:int = -1;
    public var playerName:String = "";

    public function MoosesPlayerEvent(type:String, playerId:int, playerName:
        String, bubbles:Boolean=false, cancelable:Boolean=false)
    {
      super(type, bubbles, cancelable);

      this.playerId = playerId;
      this.playerName = playerName;

    }

    public override function clone():Event
    {
      return new MoosesPlayerEvent(type, playerId, playerName, bubbles,
          cancelable);
    }

    public override function toString():String
    {
      return formatToString("MoosesPlayerEvent", "playerId", "playerName", "type
          ", "bubbles", "cancelable", "eventPhase");
    }

  }

}
```

MoosesUnit.as

```
ï»¿package  no.ekse.mooses
{

  /**
   * ...
   * @author Magnus Ekse
   */
  public interface MoosesUnit
  {
    function keyDown(keyId:int):void;
    function keyUp(keyId:int):void;
    function getMoosesPlayer():MoosesPlayer;
    function setMoosesPlayer(player:MoosesPlayer):void;
  }

}
```

APPENDIX

E

QUESTIONNAIRE

Below are the questions from the questionnaire used to gather information during the playtests.

About you

Gender
Female Male

Age
16 or younger 17–19 20–22 22–25 25–30 30 or older

Hours spent on video games per week
0–2 3–6 6–9 More than 9

How often do you use a mobile phone?
Never Rarely Regularly Often Very often

How often do you use a mobile phone for playing games?
Never Rarely Regularly Often Very often

Your gaming experience

How many played with you?
2–3 4–5 6–8 9–11 12 or more

It is easy to use the mobile phone as a gaming controller.
Strongly disagree Disagree No opinion Agree Strongly agree

It is an advantage to use your own mobile phone as a controller.
Strongly disagree Disagree No opinion Agree Strongly agree

Many people playing at the same time gives a unique gaming experience.
Strongly disagree Disagree No opinion Agree Strongly agree

Every player being in the same location increases the social value.
Strongly disagree Disagree No opinion Agree Strongly agree

It is an advantage that everyone is using the same screen.
Strongly disagree Disagree No opinion Agree Strongly agree

The games suit all players, independent of age and gender.
Strongly disagree Disagree No opinion Agree Strongly agree

Technology potential

I am willing to pay in order to play like this, given it is possible to win a prize.
Strongly disagree Disagree No opinion Agree Strongly agree

It is better to play this sort of games with product placement than watching commercials before
Strongly disagree Disagree No opinion Agree Strongly agree

ProductBall

The game is fun.
Strongly disagree Disagree No opinion Agree Strongly agree

I easily understood the objective in the game.
Strongly disagree Disagree No opinion Agree Strongly agree

I felt my vote contributed to the success of the team.
Strongly disagree Disagree No opinion Agree Strongly agree

Working together by voting to control a playable character worked well.
Strongly disagree Disagree No opinion Agree Strongly agree

I felt teamwork was important for a good result.
Strongly disagree Disagree No opinion Agree Strongly agree

I felt the game promoted teamwork well.
Strongly disagree Disagree No opinion Agree Strongly agree

Achtung

The game is fun.
Strongly disagree Disagree No opinion Agree Strongly agree

I easily understood the objective in the game.
Strongly disagree Disagree No opinion Agree Strongly agree

I had no problems finding my player on the screen.
Strongly disagree Disagree No opinion Agree Strongly agree

I easily understood how to control the player.
Strongly disagree Disagree No opinion Agree Strongly agree

TowerDefense

My role in the game.
Attacker Defender

The game is fun.
Strongly disagree Disagree No opinion Agree Strongly agree

I easily understood the objective in the game.
Strongly disagree Disagree No opinion Agree Strongly agree

I had no problems finding my player on the screen.
Strongly disagree Disagree No opinion Agree Strongly agree

It is easy to see my team's progress in the game.
Strongly disagree Disagree No opinion Agree Strongly agree

I felt the team could work together in order to achieve its objective.
Strongly disagree Disagree No opinion Agree Strongly agree

I felt the game promoted teamwork well.
Strongly disagree Disagree No opinion Agree Strongly agree

I felt teamwork was important was important for a good result.
Strongly disagree Disagree No opinion Agree Strongly agree

I felt I could contribute to the success of the team.
Strongly disagree Disagree No opinion Agree Strongly agree

BIBLIOGRAPHY

[Ado] Adobe, *Adobe flash player pc penetration - accessed october 20th 2009*, `http://www.adobe.com/products/player_census/flashplayer/PC.html`.

[Arm] ArmorGames, *Gemcraft - accessed december 10th 2009*, `http://armorgames.com/play/1716/gemcraft`.

[Des] Amanita Design, *Machinarium - accessed december 10th 2009*, `http://machinarium.net`.

[Føl09] Esben André Føllesdal, *Implementation and evaluation of selfish - a social game for mooses*, Master's thesis, Norwegian University of Science and Technology, June 2009.

[HDRS93] Victor R. Basili H. Dieter Rombach and Richard W. Selby, *Experimental software engineering issues - critical assesment and future directions*, 1993, pp. 3–12.

[Heg08] Vebjørn Heggdal, *Microsoft xna game development for multiplayer on one screen entertainment system*, Master's thesis, Norwegian University of Science and Technology, June 2008.

[Kva07] Audun Kvasbø, *Mooses game concept*, Master's thesis, Norwegian University of Science and Technology, June 2007.

[Mor07] Sverre Morka, *A flexible client for multiplayer on one screen entertainment system (mooses)*, Master's thesis, Norwegian University of Science and Technology, June 2007.

[Nin] NinjaWiki, *Bloons - accessed december 10th 2009*, `http://www.ninjakiwi.com/Games/Bloons-Games/Play/Bloons.html`.

[Pro] ProjectorGames, *Projectorgames website - accessed november 20th 2009*, `http://www.projectorgames.net`.

[RJCW] Omar Rodriguez, Erik J. Johnson, Scott Crabtree, and Brad Werth, *Carry small, game large: Big shared screen multiplayer gaming - accessed november 20th 2009*, `http://www.gamasutra.com/view/feature/3649/carry_small_game_large_big_.php`.

[SV07] Aleksander Baumann Spro and Morten Versvik, *Multiplayer on one screen entertainment system*, Master's thesis, Norwegian University of Science and Technology, July 2007.

[Wika] Wikipedia.org, *Adobe flash - accessed september 7th 2009*, `http://en.wikipedia.org/wiki/Adobe_flash`.

[Wikb] _____, *Adobe flex - accessed september 8th 2009*, `http://en.wikipedia.org/wiki/Adobe_Flex`.

[Wikc] _____, *Video games - accessed november 26th 2009*, `http://en.wikipedia.org/wiki/Video_games`.