Abstract

This master thesis presents an approach to make the MOOSES platform more flexible and dynamic. MOOSES is a platform that allows people to play video games on a large screen using their mobile phones. A server runs the game on the large screen, and mobile phones are used as controllers for the game. The client controller is an application written in Java 2 Micro Edition (J2ME), that has to be downloaded to the mobile phones. The game server communicates with the client controllers through bluetooth. The MOOSES prototype was developed during our depth study at NTNU fall 2006 based on ideas from us and Alf Inge Wang.

During the development of the MOOSES framework we encountered some technical limitations that constrained the MOOSES platform. One of these problems was tied to maintenance of the platform that makes adding and removing of games to a bothersome job.

Software development on mobile devices has become more important over the last years, and new solutions have been made to program on this platform. Most mobile phones support the J2ME platform, and there has been developed a couple of script solutions that run on top of this platform. Scripts provide for the J2ME application programming interface (API) to be more powerful by elevating the abstraction level. It also make it easier to provide domain specific set of instructions, making development easier.

The MOOSES prototype requires manipulation of the source code when adding new games. As a consequence of this, a new client has to be distributed to all the users of the system every time new games are added. A more dynamic client will provide for the MOOSES system to be passive in the addition of new games because the games will not interfere directly with the backbone of MOOSES, but are loaded dynamically on runtime.

We will also focus on making development of game clients more tailored to the MOOSES concept. When we are using a scripting platform to develop game clients, it will provide an API that is completely tied to the MOOSES platform. Such an API will have both pros and cons. It makes the API easier to follow and provides a better overview, but it constrains the freedom of creativity.

We will have a look on different scripting solutions that may replace or stand as an alternative to our current solution for development of game clients. We will also develop a prototype to test the feasibility of such a solution.

A client for the game developed in the depth study will be rewritten to suit the scricted client prototype. This client will be tested at our presentation of MOOSES at the Kosmorama film festival 2007 [1].

Preface

This master thesis report is written by Sverre Morka in the period from January 2007 to June 2007.

Acknowledgments

We would like to thank Alf Inge for this assignment and for his help and guidance throughout this project. We will also like to thank Metallica, Rammstein, Ayreon and Hans Zimmer for making this project easier providing good music making this research much more enjoyable.

We will like to thank Tellu for making ServiceFrame, and for showing great involvement in our project and taking care of public promotion of our concept.

The MOOSES crew will also like to thank everyone that showed up to test our product at the Kosmorama film festival, and all the journalists that gave us positive feedbacks.

We would like to thank all the developers of the Script languages that are discussed in this report, and above all David N. Welton and Wolfgang Kechel for developing Hecl that are used for the prototype.

Last but not least we will thank Herman Friele for making great coffee that has been crucial to upheld the motivation in dark times.

Trondheim, 15. June 2007

Sverre Morka

Contents

List of Figures

List of Tables

Listings

Part I

Introduction

CHAPTER 1

Introduction

Multiplayer On One Screen Entertainment System (MOOSES) is a platform that allows people to play multiplayer games on a projected screen, using their mobile phones as controller for the game. The client that allows the mobile phone to function as a controller is written in Java 2 Micro Edition, and has to be downloaded and installed to the client. The game runs on a server which communicates with the mobile phones using Bluetooth 2.0. The MOOSES prototype was developed during our depth-study fall 2006 at NTNU.

MOOSES has been given great response from the audience whenever we have tested it or published information about it. The enthusiasm we have met has been better than we ever dared to hope for. The presentations done so far have been executed with use of the prototype we developed during our project last semester. This prototype still lacks a lot of elements to make it a complete, reliable and user friendly platform.

This project is one of two projects that aims to take MOOSES into further development to increase it's capability to meet its goals. During this master thesis we will try to cover some of the fields of the Further work section in our depth study that we found most important. These fields are:

- ➢ On demand downloadable games for the MOOSES platform
- ➢ Look into what kind of GUI (Graphical User Interface) options that should be available in the library for gaming.
- ➢ Better solutions for loading different game clients for the J2ME (Java 2 Micro Edition) client until class loading becomes viable, or enable scripting.
- ➢ Scalability testing and improvements

- Look at interoperability between different types of mobiles, find minimum requirements.

Some of these fields will be given more attention and be prioritized higher than others. This projects main focus is development environment of the game clients used by the MOOSES platform, and testing will be included as a natural consequence of this development. Looking at interoperability between different types of mobiles will in this turn be limited to the number of different mobile devices available during testing.

1.1 Motivation

The MOOSES prototype developed in our depth study was used as a proof of concept during testing. The client side of MOOSES is based on the Java 2 Micro Edition (J2ME), which is the most common development platform for mobile devices [2]. J2ME require the classes that constitute an application to be pre-verified, and data to describe the application to be stored in a separate file [3]. As a consequence of this, applications developed on the J2ME platform may not be updated by adding or replacing classes. An update of the application requires the whole jar file that contains the classes constituting the application to be replaced.

For MOOSES this means that for every game added to a MOOSES implementation, the mobile client has to be rewritten, preverified, compiled, and redistributed to all the clients.

This results in redistribution of the client for every game update at the gameserver and causes unecessary traffic on the bluteooth bandwidth. Also, having to go through the edit-verify-compile process is bothersome compared to just adding the games without having to manipulate the server or the client.

We want to look into opportunities to make the MOOSES platform more flexible. It is crucial that the MOOSE System is able to dynamicly handle games added and games removed to provide for the customers of the system to control which games that are installed with their implementation.

Ideally the client should only be downloaded once to the participants mobile devices, and dynamicly load the games from the server. The client should only be replaced at the participants mobile device whenever the client version is updated.

1.2 Problem Definition

This project will focus on research on methods to bypass J2ME's lacking support for dynamic changes to the applications. The MOOSES platform is not able to update the clients with new controllers for new games because of J2ME's lack of a dynamic classloader. However, application ressources may be changed and added to the J2ME applications dynamicly during runtime. If we can use a resource to define the application, for instance a script, we might be able to load the game controllers on the client dynamically.

This approach means that the game controllers has to be written as scripts. And we will therefore have a look on how a scripting approach will change the game development and which utilities a scripting platform crave for development of game controllers for MOOSES.

We will make a prototype to test the MOOSES platform with a scripted client. The solution will be based on some kind of script/interpreter middleware, and research will focus on what is needed from such a solution.

1.3 Project Context

This project is a part of the research program on videogames at The Norwegian University of Science and Technology (NTNU) [4]. This research program has the following superior goals:

1. To be initiator for new research projects regarding videogames at NTNU
2. Work to identify potential external resources that may support NTNU's activities regarding videogames.
3. Coordinate and arrange for research regarding video games internally at NTNU and between NTNU and external collaborators.
4. Work to promote and make visibe the activities regarding videogames at NTNU.
5. Serve as advisors with respect to drawing up study offers related to videogames at NTNU.

The project runs in paralell to two other projects based on the MOOSES platform. Morten Versvik and Aleksander Baumann Spro have a project that focus on co-operative games and social aspects that the MOOSES platform should have [5].

Audun Kvasbø also works on new game concepts to MOOSES [6].

1.4 Readers Guide

The contents of this report vary with respect to focus and approach. Some of the report might be more interesting to some readers than others. In order to increase the readability of this report we will present a brief outline of the different parts.

If you are not interested in reading the whole report, you should identify yourself with one or more of the following categories:

Readers interested in scripting technologies for mobile devices - Should read Chapter 8, Chapter 10, Chapter 14 and Chapter 19.

Readers interested in the MOOSES concept - Should read Chapter 2 and Chapter A.

Readers interested in the technologies that the MOOSES platform is built on - Should read Chapter 6, Chapter 7 and Part III which reflect the MOOSES architecture.

Developers interested in developing games for MOOSES - Should read and understand Chapter 12, Chapter 19 and Chapter B.

Developers interested in improving the scripted client - Should read Part III to get an understanding of the architecture, Part IV to get an understanding of how things are implemented and the Part V to get a reflection of the bottlenecks.

1.4.1 Part description

In this section we will give an overview of the contents of the different parts of this report.

Part I Introduction - Gives an introduction to the purpose of this project and an introduction of the MOOSES system. This part will also include the research questions we seek to answer during this master thesis.

Part II Prestudy - The prestudy comprise study on technologies and COTS that we will make use of to answer the research questions and to build a prototype.

Part III The MOOSES Architecture - This parts presents the architecture for the prototype of the scripted client. This includes the requirements and the design decisions, together with the stakeholders interests.

Part IV Development - This part comprise notes from the development process and technical information about the implementation.

Part V Testing - This part presents the testing results, and the refection of the prototypes satisfaction of the requirements.

Part VI Discussion and Conclusion - This part contains the discussion, conclusion and answers to research questions, and further work that has to be done to the scripting platform.

Part VII Appendix - This part contains a list of abbreviations used in this report, examples of games developed on the scripting prototype and the grammars of the scripting platform.

Bibliography - This part contains the references on sources used in this project

CHAPTER 2

Multiplayer On One Screen Entertainment System

This chapter will give a brief presentation of the MOOSES platform to provide for the reader to familiarize with it.

2.1 MOOSE - System

The MOOSE system was created during the depth study done by Morten Versvik, Aleksander Spro and myself, Sverre Morka fall 2006 at NTNU. In the previous project we referred to the system as MOOS, but we changed the name of the concept to include an E and S at the end to make a catchier name which is easier for people to remember. The new name of the concept is, as the title of this section implies, MOOSE system or MOOSES. Figure 2.1 shows a physical view of the MOOSES platform.

MOOSES is a prototype that lets many people in a cinema auditorium play video games using their mobile phones as controller. The concept was originally intended to be used on the Sony 4k projector at Nova cinema in Trondheim, but has proved to fit other HD-projectors with lower resolutions. The games are designed to place multiple players on one screen, keeping all the players visible at all times. The players control their character or characters through a mobile phone with an installed application that communicate with the game server through Bluetooth. The game server is responsible for running the game and send signals to a projector that displays the game on a canvas.

The first game we developed for MOOSES was called SlagMark. SlagMark is based on ideas from the old game worms. Every player controls a worm, and the goal of the game is to kill as many of the other worms as possible. The worms kill each other by using weapons provided on

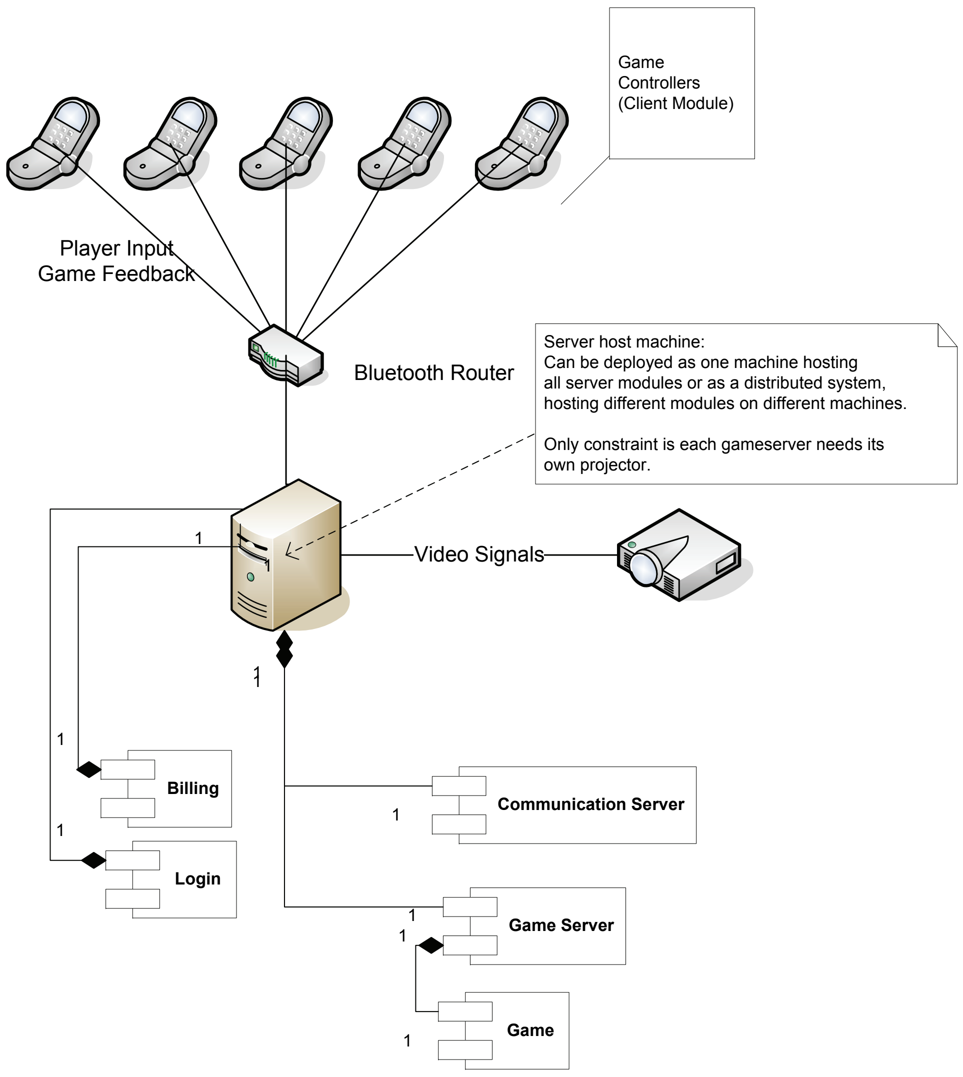
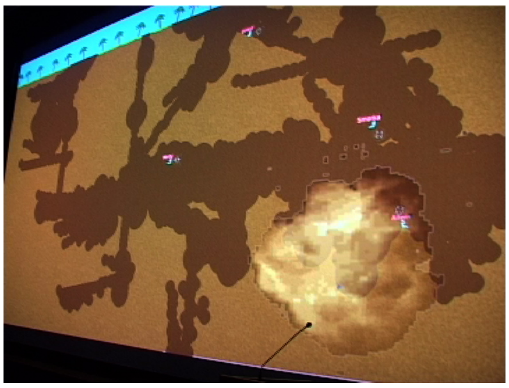
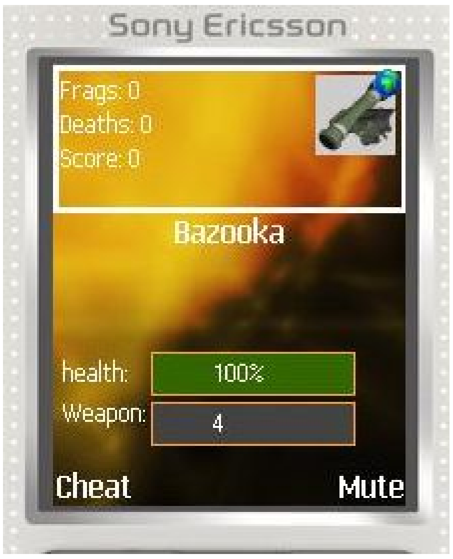

Game Controllers (Client Module)
Player Input
Game Feedback
Bluetooth Router
Server host machine:
Can be deployed as one machine hosting all server modules or as a distributed system, hosting different modules on different machines.
Only constraint is each gameserver needs its own projector.
Video Signals
1
1
1
1
1
Billing
Login
Communication Server
Game Server
Game
1
1
1
1

Figure 2.1: Physical view of the MOOSE system

the game client controller. Screenshots from the first MOOSES game can be found in Figure 2.2.


(a) MOOS Extermination game



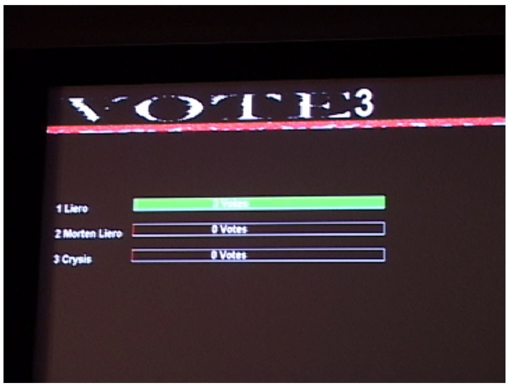
(b) MOOS Extermination controller

Figure 2.2: Screenshots from our test game

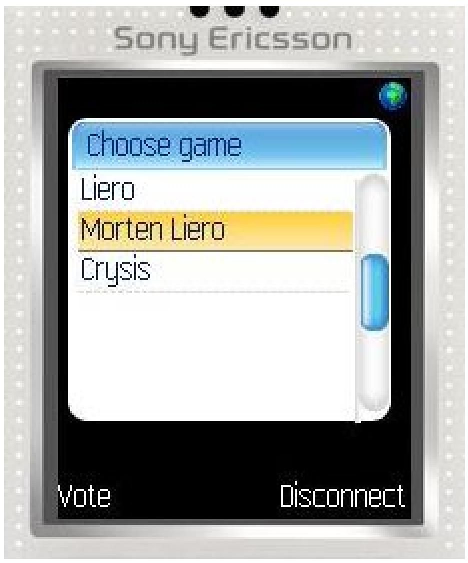
The MOOSE framework supports sound feedback both from the game and the client controller.

The game server will have multiple games installed, there is no upper threshold for maximum is set at this time. The concepts we had in focus, and the test game we developed, in the first turn were instant action games that were on for about five minutes before they restarted. Games installed on the MOOSE platform will be based on a window of time to run. When a game is ended, the highscore is displayed. Further, the server initiates a voting session, providing all the alternatives of installed games to the clients. The clients vote for the game they want to start, and the game with most votes is started. The server tells the clients to load the controller canvas to the corresponding game. At this point, all the games controller canvases are part of the client applications source code. Figure 2.3 shows screenshots of the voting part of the MOOSES. Communication between the server and the mobile clients makes use of Bluetooth 2.0 technology.

Additional information on MOOSES can be found in the depth-study report [7] or at our homepage www.mooses.no [8]. Videos of the first MOOSES test session, and presentation for the norwegian television show Newton may be found on the disc bundled with this report.



(a) Vote screen



(b) Vote controller

Figure 2.3: Screenshots from the vote session

2.2 Development platform

The MOOSES platform depends on developers to develop and supply games. To attract developers to our concept we would need a development platform and some standardizations for common game genre profiles. The game development to our concept as it is today requires the developer to make both the game in c++, Java or any other language that supports the Java Native interface at the server side, and a client class in Java 2 Micro Edition.

The java part on the client side causes a problem because of the lack of Java's dynamic class loader support. To develop a client for your game to the MOOSES platform prototype, you got to make one or more java classes that defines canvases with methods for:

- ➢ Graphical representation
- ➢ Animations
- ➢ Data exchange with the server
- ➢ Response for user input
- ➢ Response for game data received from the server
- ➢ Game logic
- ➢ Switching between canvases

The author will also have to define message classes for the data exchange between the game and the client. The MOOSES framework is based on the ServiceFrame architecture developed by Tellu [9], using the Server and the Client as two different actors. After the class is edited and compiled, the client actor has to be updated with the new class and actions to the new messages created. More information on how to develop game for the previous prototype of MOOSES can be found in Chapter 11.

If the J2ME platform had supported dynamic class loading this would have worked well. Whenever a game is added, all the client would have to do is to download the new classes and load them in to the application dynamically. Unfortunately, this is not the case.

2.2.1 Alternatives to user defined class loader

To get around J2ME's lack of a user defined class loader we would obviously have to make a standardization for the data exchange, so that neither the client actor or the game server actor would need any modifications for new games, at least not for common game concepts.

A scripted solution for games may offer us an implementation that does not rely on new Java classes to define the game clients. If we provide the client with classes with abstract methods that can be accessed by a script, the client would only have to download the script and the corresponding media files (sounds and pictures etc.) whenever a game is added. However, a scripting approach might not be the ultimate solution. We will have a look on how the script solution may constrain the possibilities we already have, and compare this up to the benefits we are gaining. The utility that we expect to be most constrained by the usage of scripts are the ability to make and manipulate graphics and animations.

All in all a development platform should support easy layout for the common graphical units, and support for user input and data exchange with the game, as well as actions taken based on game data.

2.2.2 Development Tool

If the scripted implementation proves to be a success, we should build an editor tool that makes client development easier. Such a tool should include support for authoring scripts, draw the GUI and test the application. Since the scripts are stored in plain text files or analogous, they may be authored from an arbitrary text editor, however it would be more convenient to make a tool for authoring to highlight keywords etc. Such a tool would also reflect warnings and errors, making debugging much easier. We could also include functionality for script generation. The tool could provide an interface for the game client designer to draw the GUI/layout and get the script needed for these drawings generated automatically. The tool should also have an interface for manipulation of the attributes generated. A development tool should also provide functionality to test and profile the application with a simulated server.

This master thesis will focus on development of a scripting platform for the MOOSES client, and any development tool software will be developed at a later stage if the scripted solution proves to fit the concept.

CHAPTER 3

Research Questions & Methods

In this chapter we will describe which questions we seek to answer through our work, and how we intend to find the answers. We will describe how we plan to conduct the work and explain the different methodologies we plan to follow.

3.1 Research questions

This research is meant to extend the MOOSES framework to with respect to covering unsolved problems for dynamic adding and removing of games, and to provide better development environment to the game authors.

The larges barrier for the MOOSES concept is the utility to dynamically update the mobile client without having to make changes to the source code and without having to reinstall it on the mobile phones. The solution for developing game clients with the prototype developed during our depth-study has led us partly where we want to go. It provides support for balancing gameplay between the Main game and the client display in form of visual representations, animations, user input and data exchange with the game. We want to keep these utilities in the development of a new dynamic solution for the game client implementation.

Creating a dynamic way to implement the client in form of scripts require profiles on different game concepts to provide a better abstraction for game developers. Different game concepts have different needs for utilities provided by the client, and grouping them into profiles will make development environment more orderly. The research will therefore focus on suitable game concepts and which features they crave from a scrpting platform.

The current implementation requires the developer to make new messages in form of Java classes for data exchange between the game and the controller that are tailored to their game. An abstraction for game developers need to make standardization on the data flow between the game and the client to prevent manipulation of the MOOSES source code.

We want to provide some research into determining solutions to come around the lack of a user defined class loader for J2ME. The Connection Limited Device Configuration for J2ME, which is the configuration used by mobile phones, does not support dynamic user defined class loading. This is constrain the MOOSES opportunities to load new game controllers dynamically.

Alternative solutions may constrain and put limitations on the former implementation. Some of our concept ideas from the previous project are likely to suffer some reduced opportunities. Alternative solutions might also put constraints on the developers' creativity.

In addition to game development, the alternative solutions may also suffer some technical difficulties. Using a middleware solution may produce a significant overhead, more power consumption and so on. Communication may also represent a problem, not only because of overhead produced by middleware, but freedom of creation from the developers' side.

In retrospect of this as well as from motivation and the problem description, several concrete questions have arisen. This report will lead to answering these questions.

RQ-I: **How will an alternative solution affect MOOSES?**

➢ We will have a look on different solutions to use script or interpreters to provide dynamic code for the game controller. This research will determine which attributes needed by such a solution to suit as a replacement for the current solution with respect to pros and cons.

a) Will a scripting API for the client make the MOOSES able to dynamically load new games?

b) To what extent will the performance of MOOSES be affected by a scripted client?

c) Will the MOOSES concept be constrained by a scripted client?

RQ-II: **Which impact will a scripting platform have on game development?**

➢ To provide a better development platform for the game developers we would need an abstraction from the current implementation. The developer should only be concerned about the response to user input, dataflow to the game, actions taken by game data and visualization. The developer should not be concerned by the background logic for making this possible.

a) Will a scripting platform make it easier to develop a game client?

b) Will a scripting platform constrain the game clients with respect to game play?

c) Which elements would a scripting API need for game client development?

3.2 Research Method

In order to make prescriptions about the research methodology in software engineering we need a basic understanding of what software engineering is.

Basili in [10] defines software engineering as follows:

> "[...] can be defined as the disciplined development and evolution of software systems based upon a set of principles, technologies and processes."

With that in mind we can start to look at what models to use for research into software engineering. One issue with attaining good models for this type of research is that software engineering is still fairly new discipline in a scientific perspective. Unlike other sciences, models for components like processes and resources have been neglected so far even though a lot of research is going on in this field. Basili [10] does however describe three experimental models for use in software engineering research. The variations between the models are small, but focus on different areas and are parts of two distinct paradigms; the scientific- and the analytical paradigm. First model consists taking on an engineering approach, the second an empirical approach while the latter takes on a mathematical approach.

The three approaches in short:

The engineering approach (scientific) In this approach one has to perform iterations of observing the existing system, suggesting improvements and building and analyzing the new system. This continues until no more improvements can be found.

The approach is strictly evolutionary and implies access to existing models of processes, products and the environment in which the software is developed.

The empirical approach (scientific) Based on a model of the domain a set of statistical and qualitative methods are proposed. Then these models are applied to case studies, measured and analyzed, and the result is a validation of the model.

This distinct the approach from the previous one since a new model is proposed. It is also more reliable to validate the model through the use of case studies. This approach is widely used in all fields of research.

The mathematical approach (analytical) A formal theory or a set of axioms is presented, the theory is developed and a result is derived from it. It is preferable to have this results compared to empirical observations.

The empirical approach will constitute the base research method for resolving the game related research questions to determine the utilities a scripted solution have to support for the different games, while the engineering approach will constitute most of the work behind the extension of the framework.

3.2.1 The Engineering Approach

In Section 3.2 above, we described the engineering approach as; observing, suggesting improvements, analyzing and building. The observation phase consists of the Part II Prestudy where we will look into game concepts to profile them and aspects of the scripted solution as well as some technology used by a scripting approach.

With the main focus of this project being to design a script platform for use of game clients on the MOOSES concept, we will incorporate the last two stages of the engineering approach into the requirements, design Chapters of Part III Architecture and development Chapter of Part IV Development of the scripting Framework. The last part of the engineering approach, suggesting improvements, will be found Chapter 25.

3.2.2 The Empirical Approach

As mentioned in Section 3.2, we will use the empirical approach to research different game concepts in order to determine what they will require from a scripted solution, and it they will fit a scripted solution. We will use an adapted form of literature study to categorize the games we found feasible in the first project. With this data we can determine what elements will work with a scripted solution, and which constraints such a solution may put on them. This will also help in spite ideas for game concepts for our domain.

We have chosen to categorize the different game types into the following sections:

Genre presentation will try to summarize the common basic gaming elements the given genre makes use of.

Feedback will try to summarize the most common information that is necessary to reflect to the user.

Visualization will try to find common visualization techniques and how to reflect them on the mobile.

When the detailed look at the game types is done we will summarize the elements needed by a scripted client, and some possible loopholes which one should look out for. We will also develop a prototype to test a scripted client implementation on the games we have so far.

3.3 Test Environment

The testing will reveal if our solution was satisfactory. The testing will be performed on the emulators on desktop computers and on mobile phones. The framework will be tested at the main auditorium at Nova with real cellular phones at the Kosmorama film festival in Trondheim. This to make sure the framework performs acceptably in an optimal setting.

CHAPTER 4

Development Tools & Software

In this chapter we will give a brief presentation of the different tools used to create this report and the prototype.

4.1 Eclipse With Plugins

Eclipse [11] is a an open source development platform with lots of available plugins for different purposes, e.g. writing in LaTeX, creating applications for mobile phones and finding code statistics.

TeXlipse is a plugin for writing LaTeX documents in Eclipse. It includes features like syntax highlighting, command completion and bibliography completion [12].

EclipseMe [13] is a plugin for developing J2ME MIDlets in Eclipse. A Java Wireless Toolkit must be installed on the system for the plugin to work.

4.1.1 MiKTeX 2.5

MiKTeX is an implementation of TeX and related programs for Windows on x86 systems [14]. The MiKTeX distribution contains many features including the pdfTeX compiler which generates a pdf document. The compiler is used to produce this document.

4.1.2 Concurrent Version System

Concurrent Versioning System (CVS) are used to keep track of versions of code and documentation. In our case we will use the CVS to keep backups and make the project available from multiple terminals. NTNU has a Linux server with CVS support which we have used in this master thesis. The system also makes it easy to work from any computer that has Eclipse and the necessary plugins.

4.2 Emulator

Applications made in J2ME need to be tested. The applications can run on mobile phones, but it is a tedious process to do for each iteration. Therefore, emulators can be used instead. There are several emulators available and each mobile phone producer has its' own toolkit. In our study we will use the Sony Ericsson SDK.

4.2.1 Sony Ericsson SDK

The Sony Ericsson software development kit (SDK) is a wireless toolkit that can be used to emulate Sony Ericsson mobile phones that support Java ME technology [15]. The kit can emulate the following phones: W800, W600, W550, Z520, K750, K600, K300, J300, Z800, V800, S700/S710, Z500, K700, Z1010, K500, K508, F500i, P900, P910, Z600/Z608, T630-T628, T637 and T610 Series (T610, T616 and T618). The SDK can run applications that uses both MIDP 1.0 and 2.0.

The Sony Ericsson toolkit can run several instances of an application using emulations of different phones so that each phone has a recordstore, filesystem and PIM database. This enables us to test the applications with the emulator, without having to use actual phones. However, using this SDK will only test how the framework and applications will work on Sony Ericsson mobile phones. Therefore, the framework must be tested on different real phones ensure that the software will run on different hardware. This SDK also include profilers for communication, CPU instruction cycles and memory usage. We will make use of the CPU instruction cycles profiler during development and testing.

Part II

Prestudy

CHAPTER 5

Prestudy Introduction

This part presents the prestudy of our project. We will have a look on the tecologies that the MOOSES client make use of, and how the may be affected or contribute to the scripting platform for gamedevelopment. The tecnologies presented are Java 2 Micro Edition (J2ME) and ServiceFrame. J2ME, presented in Chapter 6, is the base of the client application and it is also the source of the problem with dynamic addition and removing of games to the MOOSES implementations. ServiceFrame, which is a framework that we have based the entire MOOSES platform on, is presented in Chapter 7.

In addition to these technologies, we will have a look on scripting technologies and central concepts regarding use of scripts. We will look on available open scripting platforms for J2ME to find one that suits our needs, which we will make use of to develop the prototype. We will also provide a detailed presentation of this language's architecture and demonstration of usage. We will also have a look at the threats that are represented by implementation of scripts on a mobile platform.

The prestudy also contains research in elements that have to be provided to a development-platform that make use of scripts. In Chapter 12 we discuss the elements that we find appropriate for the game genres that suit the MOOSES platform. In Chapter 11 we will have a look on how game clients are developed on the first MOOSES implementation to get some ideas to elements needed. We will also have a brief look on new game concepts that makes impact on how to think when developing game clients for MOOSES in Chapter 13.

CHAPTER 6

Java 2 Micro edition

In this chapter we will have a look on the Java 2 Micro Edition (J2ME) which the MOOSES client is based on. J2ME is the source of the problem taken into account for this project. We will have a brief look on the J2ME architecture and the class loader implementation for J2ME.

Java Platform, Micro Edition is the most ubiquitous application platform for mobile devices across the globe [2]. Sun Micro systems introduced a Java 2 platform called Micro edition as a set of specifications to pertain to Java on small devices to meet the increasing technology in this field with a standardized platform [3].

The domain of J2ME are ranging from pagers and mobile phones to set top boxes and car navigation systems. The J2ME architecture is divided into three layers as shown in figure 6.

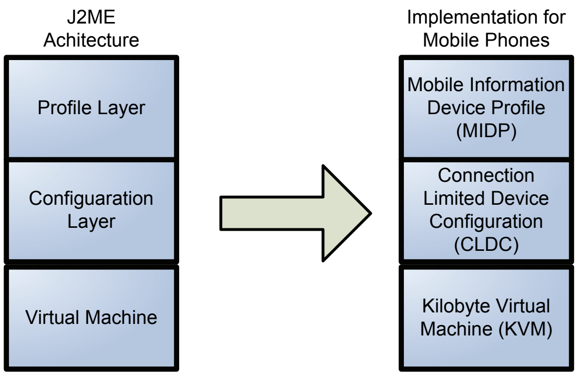

J2ME Achitecture
Profile Layer
Configuaration Layer
Virtual Machine
Implementation for Mobile Phones
Mobile Information Device Profile (MIDP)
Connection Limited Device Configuration (CLDC)
Kilobyte Virtual Machine (KVM)

Figure 6.1: J2ME architecture

6.1 Virtual Machine

The virtual machine layer implements a Java virtual machine customized for the device's host operating system that supports a particular J2ME configuration. For mobile phones this virtual machine is called Kilobyte virtual machine. During the development of the KVM, classes that were too bloated or not as critical to the system where extracted. Most important of these features, regarding our project, where the support for user-defined class loaders.

6.2 Configuration Layer

The configuration for J2ME devices defines a minimum set of Java Virtual machine features and core Java class libraries available to a category of devices. The configuration for J2ME devices consist of two alternatives, the Connected Device Configuration (CDC) and the Connection limited Device Configuration (CLDC), where CDC is a super set of CLDC. As the name insinuate the CDC configuration is aimed at devices that are always connected (although it is used on PDAs) but relatively resource poor, such as a satellite TV receiver or WebTV. The CLDC is the configuration used on mobile devices, such as mobile phones and pagers. The CLDC consist of a set of additional classes contained in separate package. These packages are the java.io, java.lang, javal.util and javax.microedition.io. The current CLDC version, CLDC 1.1, has been granted an additional package java.lang.ref to support weak references.

The verification technique used by CLDC had to be less memory consuming than the one used in Java 2 Standard Edition (J2SE). To reduce the client side verification overhead, CLDC uses a dual pre-verification process to push part of the verification operation into the development platform. Applications in J2ME are called MIDlets. When a MIDlet is compiled, the classes that constitute it is archived into a Java archive (JAR) file. The preverification process generates information called stackmap attributes that is stored in an independent file called Java Application Descriptor (JAD). This file is downloaded together with the jar file that constitute the application.

6.3 Profile Layer

The profile layer is more device specific and defines the minimum set of application programming interfaces available on a particular group of devices, which are developed on top of the underlying configuration. The profiles maintains device specification and device portability which assures that applications written for a particular profile should port to any device that conforms the profile. For mobile phones the current profile is Mobile Information Device Profile (MIDP) 2.0.

MIDP provides a Java Application Programming Interface (API) related to interface, persistence storage, networking and application model. The platform for mobile Java applications provides a mechanism for MIDP applications to persistently store data across multiple invocations. This persistent storage mechanism can be viewed as a simple record-oriented database model and is called the record management system (RMS) [16]. MIDP also put some hardware requirements to the manufacturers. To support MIDP a mobile device has to have 128 K bytes of RAM, 8 K bytes of application-created data and 32 K bytes of Java heap. Most (if not all) Mobile devices developed today has far more resources than this. For instance one of the most advanced J2ME enabled phones, Sony Ericsson's k800i, has up to 3000 Kbytes available heap [17].

6.4 User Defined Class Loader

The dynamic loading of classes is a crucial feature of Java virtual machines. It allows the Java platform to load and install new software components at run-time.

In CLDC, the application programmer cannot override, modify, or add any classes to the protected system packages, i.e., configuration specific, profile-specific, or manufacturer-specific packages. Therefore, in order to protect system classes from downloaded MIDlets, system classes are always searched first when performing a class file lookup and the application programmers are not able to manipulate the class file lookup order in any way [18].

J2ME's CDC (Connected Device Configuration) layer is a super set of the more constrained CLDC which contains the support for user defined class loaders. The CLDC's lack of a user defined class loader is mainly for security reasons, and the versions of CLDC that are developed in the future is therefore not likely to provide this utility. The CLDC class loader is a built-in "bootstrap" class loader that cannot be overridden, replaced or reconfigured, and we will therefore have to look at other options to make a client that supports dynamic adding of new games. Compiled Java class files are stored as the machine code for the Java virtual machine, referred to as bytecode [19]. It could be possible to write an interpreter that interpreted the bytecode from the class files, but such an interpreter would probably be to complex and unstable, not to mention the overhead it would produce. There has been developed multiple solutions for scripting on top of J2ME, and many of these solutions are open, meaning that we may tailor them to suit our needs. It is more likely that a scripted solution may serve as a replacement for the dynamic class loader than to interpret the bytecodes. We will therefore look into some of these solutions.

6.5 Summary

J2ME provides an advanced API that makes it possible to develop increasingly advanced applications on a mobile platform. The new MOOSES client will still be based on J2ME, but since J2ME has a strong focus on security we will have to use some kind of middleware that can interpret structural data and instructions constituting a game client runtime.

CHAPTER 7

ServiceFrame

TellU [9] is a small company located in Asker which has developed ServiceFrame [20] which is a library for use in developing of mobile applications that make use of distributed services. This library provides functionality that differs from the general by shoving more of the work to the server, and making the mobile device more passive. ServiceFrame is able to determine which connection options to use, once the available ports has been opened.

The applications consist of so called actors, equal to the Java Beans in regular java. One actor serves a service-role, and will initiate actions from given input.

ServiceFrame was a big help for us creating the prototype of MOOSES. The ServiceFrame architecture provided a platform that minimized the work we had to do to make the communication between the server and the phones. Both the server and client where easy to develop using the state machine architecture, provided by ServiceFrame.

7.1 ActorFrame

An Actor is a composite object having a state machine (ActorSM) and an optional inner structure of Actors [21]. Some of these inner Actors are static, having the same lifetime as the enclosing Actor, and others are dynamically created and deleted during the lifetime of the enclosing Actor. The state machine of an Actor will behave according to generic actor behaviour, common to all actors, and a Role type, which is bound when the Actor is instantiated. If the Actor shall play several Roles, this is accomplished by creating several inner Actors each playing one of the desired roles.

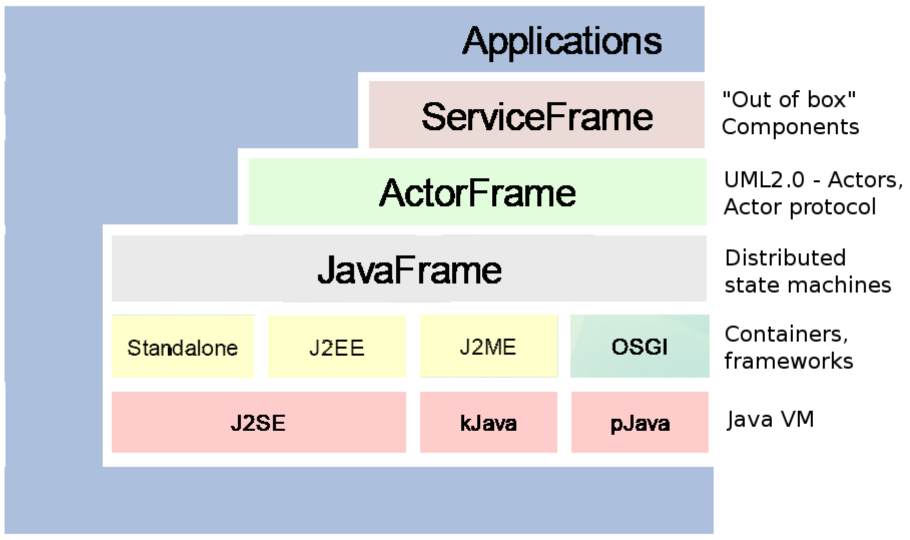


Figure 7.1: Serviceframe and actorframe overview

On the MOOSES server the server has several roles, for instance billing, highscore management, authentication, interfacing the game etc.

Communication between the Actor and its environment takes place via an in-port and an out-port. Internal communication among the inner actors is also routed via the ports.

The actor has a generic behaviour, inherited from the base Actor type that provides management functionality. It manages the inner structure of Actors and the Roles they play. It knows the available Roles and the rules for Role invocation. This is provided to it in the form of a Deployment Descriptor (DD). The generic behaviour handles role requests and will either deny the request or invoke an Actor to play the requested role or an acceptable alternative role. The generic behaviour also has the capability to add and remove roles, and to perform other Actor management functions. It keeps track of what Plays the Actor takes part in and is able to track and release all Roles in a play when the Play shall be ended.

The Actor has an assigned Role type, which defines the application specific behaviour of the Actor. The behaviour of a Role type is defined by a composite state - RoleCS.

A service designer will primarily work with RoleCS and its constituent Role features to design service behaviour.

An Actor has a unique identifier and ActorAddress. In addition, the ActorSM holds both generic attributes and application specific attributes.

ActorFrame itself is defined as a special Actor, the Root Actor. This Actor will normally have an inner structure of Actors reflecting the needs of the environment. Each of these inner Actors may recursively contain an inner structure until atomic Actors having only a role behaviour and no inner structure are reached.

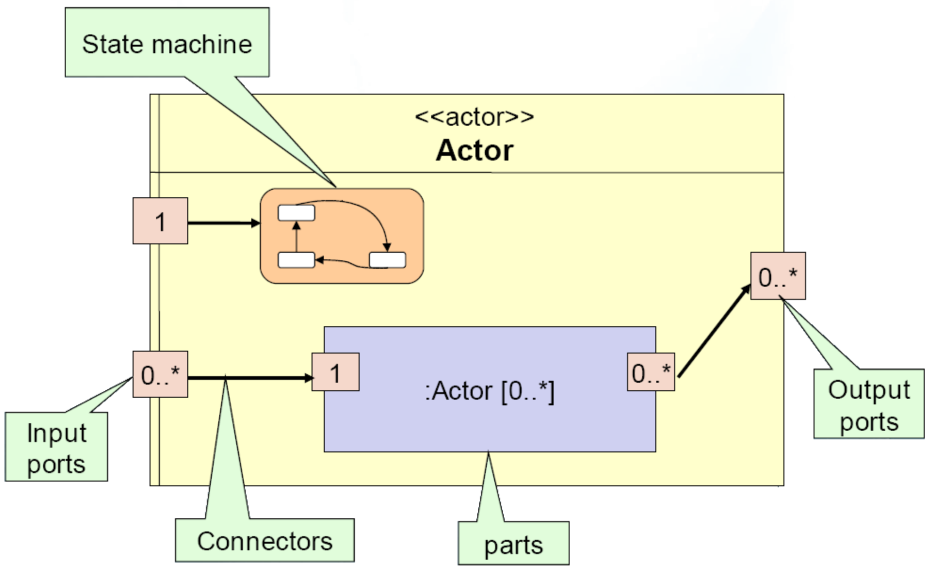


Figure 7.2: UML actor-statemachine

7.2 TellU ServiceFrame

ServiceFrame is a statemachine and message-routing framework for J2ME, J2SE and J2EE. It has support for tcp-, udp-, bluetooth- etc routers. Events are sent using messages and the routing system takes care of where the messages gets sent. ServiceFrame's generic architecture makes it very scalable.

ServiceFrame is basically a Root Actor containing an open collection of actors reflecting the needs of the application domain and a library of generic Actor types and role types (classes) tailored to the domain. This is a general framework that must be specialized by supplying application dependant Actor types and Role types and instances.

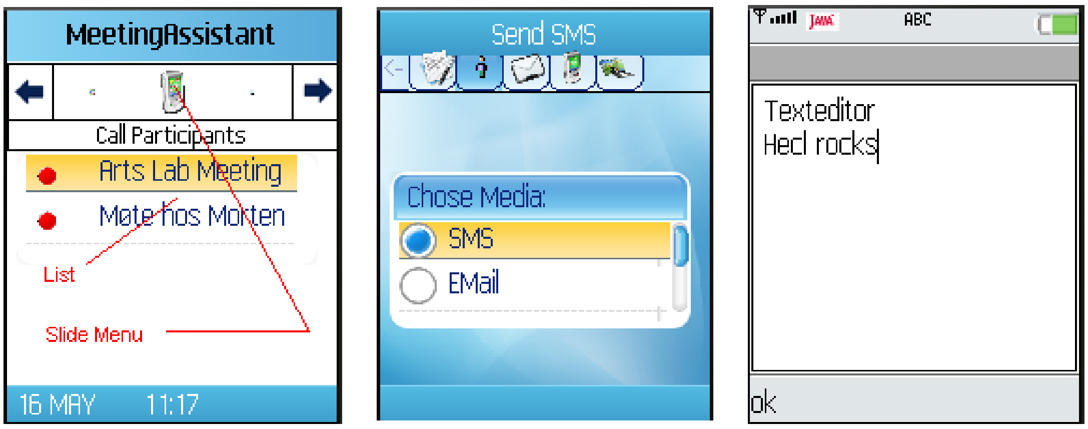
ServiceFrame allows designers to add new types and to specialize existing types incrementally. The Actors are interconnected through ports. The ports are in charge of message routing and have access to routing databases, and also the name-servers/registries needed to perform name based routing.

ServiceFrame builds on a Peer-2-Peer architecture where client server architecture is a special case. It has an event driven architecture build on asynchronous message passing and use of state machines for complex behaviour (UML based). It uses a message structure that makes application routing for a flexible integration and distribution of applications.

The ServiceFrame architecture is scalable allowing a flexible and dynamic deployment of applications making it able to support applications on large servers to pc to mobiles, to sensors. It also has integrated with external service enablers making it able to interface services(SMS, MMS, Map, Webservices).

7.3 Tellu J2ME GUI Library

The J2ME part of ServiceFrame has its own library for Graphical User Interface components. This library has support for themes, animations and common elements such as lists, text elements, menus etc. The GUI library builds on the J2ME Canvas, i.e. all the elements consists of primitives painted on the canvas. The canvases therefore supports direct drawing on the canvases, and manipulations of the elements drawing methods. Figure 7.3 shows some examples of the ServiceFrame GUI elements.



(a) Main window (b) Meeting details (c) Text editor

Figure 7.3: Service Frame GUI examples taken from a CRM application

CHAPTER 8

Script languages and grammars

In this chapter we will have a look on scripting technologies and structure to provide a better understanding for scripts purpose, architectural concerns and area of application.

8.1 Scripting Languages

Scripting languages are programming languages that can be used to write programs to control an application or class of applications [22], typically interpreted rather than compiled, and can be typed directly from the keyboard or be stored in plain text files.

Common high level programming languages convert the code into binary executable files, and yield weak support for dynamic changes once they have been compiled. Scripts remain in their original form and is interpreted command by command each time they are executed, hence a script may be typed directly from the keyboard to a command line.

8.1.1 History And Development

Scripts where created to cut down on the edit-compile-link-run process, as scripts may be edited runtime. The inventors named it script to associate with the scripts given to actors in which dialog is set down to be interpreted by actors. In a computer setting, the actors are the programs. Although scripting languages can be compiled, the common usage is interpreting, because it is easier to write a interpreter than a compiler.

8.1.2 Higher Level

Scripting languages have a higher level than system programming languages. System programming languages where made as an higher abstraction than the assembly language that consists of plain machine instructions. A statement in a system language may comprise four to six statements in assembly language. Scripts run on top of the system languages, and represent an even higher abstraction level. One statement in a script may comprise about a hundred statements in the underlying system language [23].

8.1.3 Area of application

As system programming languages are designed for building data structures and algorithms from scratch, scripting languages are designed for gluing these components together. They assume the existence of a set of powerful components and are intended primarily for connecting components together.

As an example of scripting usage we can refer to computer games. Games often use scripts to extend the game logic, tailoring the game engine to particular game data. This is convenient to make some elements accessible from within the game. In this way scripts may also make some applications programmable from within, so that repetive tasks can be quickly automated. Scripts is also used in web-development, where they may provide for the client to be less dependent on the server since the scripts allow processes to be executed locally.

8.1.4 Types And Primitives

To simplify the task of connecting components, scripting languages need to be typeless, meaning that all things look and behave the same. For example in PHP, a variable may hold a string one moment and an integer the next. This results in most scripting languages to be string oriented, since this provides a uniform representation for many different things. For instance, "hello", "123", "32.233", "true" etc. It might seem that this results in scripting languages to allow errors, however, scripting languages do their error checking at the last possible moment, when the value is used.

Scripts are typically used to control or write parts of an application rather than programming whole applications. Therefore the terms coarse grained and fine grained grammar becomes reel, and will be discussed in Section 8.3.

8.1.5 Efficiency

Scripting languages are less efficient than system programming languages. Since the scripts are being interpreted by other programs they produce a overhead, making them harder to optimize for memory management and speed. The interpretation also consume more memory than compiled applications, since the process is bigger. Also the scripts basic components are chosen for power and ease of use rather than efficient mapping onto underlying hardware. However, the script option provides an advantage in association of file size and code needed.

The performance of scripting languages is not usually a major issue because the scripted applications size are usually smaller than system language applications, with limited purposes.

8.1.6 Different types of scripting languages

To give a better understanding of the usage of the scripting languages we will present the most common types of scripting languages and give some examples on their usage. There are many different types of scripting languages based on their domain criteria, and we can divide scripting languages into following superior types [24]:

- ➢ **Job control languages and shells** - This is scripting solutions provided to control the behaviour of system programs, often written and interpreted through a command line.

  Example: Microsoft DOS's COMMAND.COM [25], used a command line interface to write DOS-commands. Alternatively they could be stored in *.bat files as a collection of DOS-commands.

- ➢ **GUI Scripting languages** - As the name states, these languages are used to control graphical user interfaces that a system generates (Windows, buttons, text fields, etc.). The support for this kind of languages are dependent on the application and the operating system. GUI scripts are often used to automate repetive actions or configure a standard state for graphical user interface.
  Example: Action script which is used to provide animations, logic and GUI components to Macromedias (Adobes) Flash [26].

- ➢ **Application-Specific Languages** - Many application programs include a scripting language that aims on the needs of the application user. This scripting type are often included in programs that provide for or require the user to make parts of the program.

  Example: Many computer games makes use of Application specific languages to define actions of non player characters and creation of own maps, weapons or gameplay elements. For instance the scripting language used in Americas Army [27] has a scripting language to make configuration details available from within the game. The commands of this language can be found at [28].

- ➢ **Web programming languages** - Application specific scripting language that is used to provide custom functionality to dynamic web pages. Modern web programming languages are powerful enough for general purpose programming. We divide these languages into two sub genres based on their purpose, namely Server-side scripting languages and client-side scripting languages.

  – **Server-side** - One subtype such as the **PHP** [29] and **SMX** [30] are used to write code for generating dynamic web pages on the server side.

  - **Client-side** - The client-side languages are used for the website to access and the client and/or execute actions on the client side rather than going through the server. For instance changes in the page layout may be coded to be executed on the client-side to minimize server traffic and latency. **Examples Javascript [31] and VBScript [32]**

- ➢ **Text processing languages** - This subtypes are among the oldest uses of scripting languages. They are designed to aid system administrators in automating tasks that involve configuration and log files.

  Examples: Perl (which has grown to be an application language) [33] and **AWK** [34]

- ➢ **General-purpose dynamic languages** - Script languages usually have a specific goal when they are created. However, some of the languages evolve to cover a broader band of purposes. For instance PHP which is intended to be a language for generating dynamic web pages may be used for programming of graphical applications.

Our script solution will have elements from both the application specific languages and the GUI Scripting languages. The language will be used to declare and control data structures that are already defined on the Java side of the framework. In addition to this, the script will also be used to control events and communication for the game client. The script is intended to provide as much functionality as the previous implementation as possible without constraining the freedom of creativity for the developers too much.

8.2 Grammars for programming languages

All languages are based on the grammar that makes the rules of how to make sentences out of words. The same goes for computer languages. In this section we will have a look on features that are element in grammars of computer languages. Grammars define both statements, expressions and tokens.

8.2.1 Lexical Analyzer

A program that accepts a sequence of characters and returns a sequence of tokens is called a lexical analyzer. A lexical analyzer is the first step in a compiler or interpreter. Once the characters are compiled to tokens it is sent to the parser. A parser is a program that accepts a sequence of tokens and returns a parse tree.

In Figure 8.1 we can se how the lexical analyzer handles a pseudo function that calculates the factorial value of an integer.

```
[fun {Fact N}
        if N==0 then 1 else N * {Fact N-1}
        end
end]
```

Listing 8.1: Factorial function example

The parse tree is provided to an interpreter that reads the tokens and executes the computations based on the language syntax.

8.2.2 Extend Backus-Naur Form

One of the most common notations for defining grammars is called Extended Backus-Naur Form (EBNF) [35]. EBNF distinguishes between terminal symbols and non-terminal symbols. A terminal symbol is a defined token, whereas a non-terminal symbol represent a sequence of tokens. The non-terminals is defined by the grammar rule which shows how it is expanded into tokens.

Example

<character> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

The declaration above says that <character> represent one of the 26 tokens (terminals) a, b, . . . , z. The symbol '|' is read as "or", and means to pick one of the alternatives. The grammar rules may refer to other nonterminals.
This nonterminal shows how to write strings

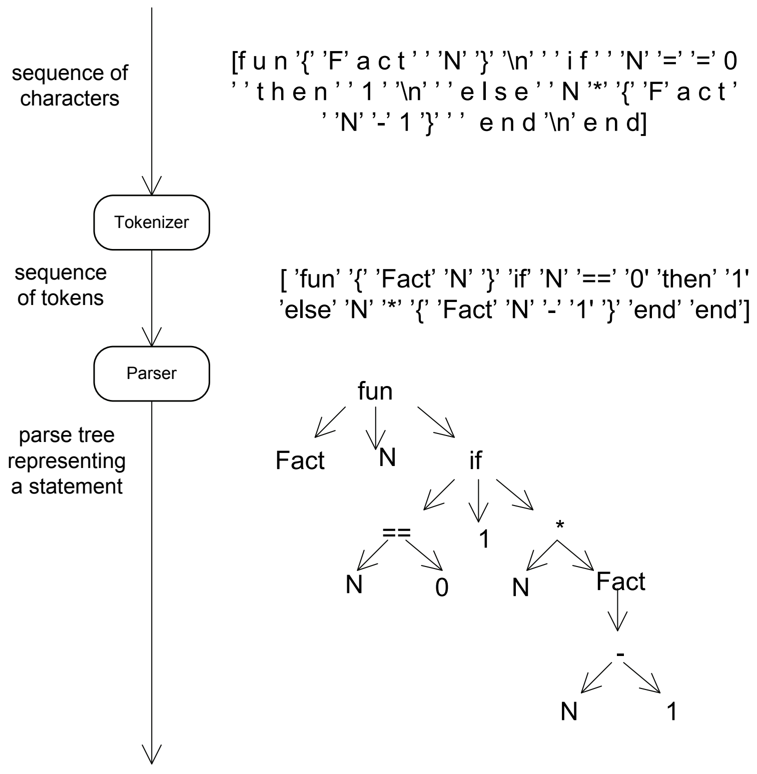

sequence of characters
[f u n '{' 'F' a c t ' ' 'N' '}' '\n' ' ' i f ' ' 'N' '=' '=' 0
' ' t h e n ' ' 1 ' '\n' ' ' e l s e ' ' N '*' '{' 'F' a c t '
' 'N' '-' 1 '}' ' ' e n d '\n' e n d]
Tokenizer
sequence of tokens
['fun' '{' 'Fact' 'N' '}' 'if' 'N' '==' '0' 'then' '1'
'else' 'N' '*' '{' 'Fact' 'N' '-' '1' '}' 'end' 'end']
Parser
parse tree representing a statement
fun
Fact
N
if
==
1
*
N
0
N
Fact
-
N
1

Figure 8.1: Lexical analyzer handles factorial function

<string> ::= <char>|{<char>}

This rule says that a string is a character followed by any number of characters. The braces mean to repeat whatever is inside any number of times including none. The braces [...] means one or zero instances and the braces {. . . }+ means one or more times.

8.2.3 Context free and Context sensitive grammars

The grammar from the example are referred to as *context-free* grammars, because the expansion of a nonterminal (<char>) is always the same independent from where it is used. The compliment to *context-free* grammars is *context-sensitive* grammars. In Context-sensitive grammars, nonterminals depend on the context where they are used. Most of the programming languages can not rely on the context-free grammar alone to define what is legal and not. For instance, most of the programming languages require a variable to be declared and often initiated before it is used. This is a context dependency.

The syntax of most practical programming languages is defined in two parts, as a context-free grammar supplemented with a set of extra conditions imposed by the language.

To prevent context-free grammars from being ambiguous they implement precedence as a condition to the parser.

Example

<expression> ::= <integer>|<Expression> <OP> <Expression>
<OP> ::= +|*
<integer> ::= 0|1|2|3|4|5|6|7|8|9

The grammar above will result in two parse trees giving different results. The expression 2+3*4 will result in either (2+3)*4=20 or 2+(3*4)=14.

Precedence is a condition on expressions with different operators. Each operator is given a precedence level. The higher the precedence, the deeper they go in the parse tree. For instance, * usually has higher precedence than +, therefore the parse tree 2+(3*4) is chosen in favour of the alternative (2+3)*4. This is expressed as * binding tighter than +.

8.2.4 Syntax Notation

Grammars are defined to make it easier for a developer to get to know and understand the language syntax and semantics. But grammars are also dependent on how the interpreter or compiler is written. The notation of the grammars often depends on how the parser is written. There are three notations used in programming languages, infix, prefix and postfix, with infix being the most common because it is more intuitive to the programmers. The three notations are different but they produce equivalent results. To demonstrate the three notation we will percent them in a list

- **Infix notation: X + Y** - Operators are written in-between their operands. This is the usual way we write expressions. An expression such as A * (B + C) / D is usually taken to mean something like: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."

  Infix notation is strictly dependent on extra information to give the order of evaluation of the operators clear. The precedence and associability rules are important. Usage of brackets ( ) allow users to override the rules.

- **Postfix notation: X Y +** - Operators are written after their operands. To demonstrate the postfix notation we can write the infix expression given above in postfix notation, A B C + * D / . Unlike Infix notation, postfix notation always perform evaluation of operators from left-to-right, and brackets can not be used to change their order. The leftmost operator is always evaluated before the next one. Because Postfix operators use values to their left, any values involving computations will already have been calculated as we go left-to-right, and so the order of evaluation of the operators is not disrupted in the same way as in Prefix expressions.

➢ **Prefix notation: + X Y** Operators are written before their operands. The expression above is equivalent to / * A + B C D . Just like postfix notation, the prefix notation always arrange the order of evaluation of operators from left-to-right, independent of brackets. Prefix expressions use values to their right, and if these values themselves involve computations then this changes the order that the operators have to be evaluated in. In the example above, although the division is the first operator on the left, it acts on the result of the multiplication, and so the multiplication has to happen before the division (and similarly the addition has to happen before the multiplication).

8.3 Coarse grained VS fine grained

If we replace the client with a script interpreter we can expect an overhead compared to running precompiled applications as the old client. Therefore it is crucial to have a look on options to reduce the overhead as much as possible. The terms coarse grained and fine grained in context of scripting languages mirrors the scripting language's capability to provide a high abstraction level on the code. As discussed in Section 8.1.2 one script statement may consist of a couple of hundred statements from the underlying system language. Such a script will be referred to as coarse grained, while a script that assembles one to three statements from the underlying language will be referred to as fine grained, because it requires most of the work to be done on the script side.

For instance, if the developer wish to invoke an animation moving a rectangle 10 pixels up , it would be easier to write:

```
animate $animation up 10
```

Listing 8.2: Coarse Grained Animation Example

than

```
set offset 50
for {set i 0 } {< $i 11} {incr $i } {
    paintrect y [- offset i] x 50 width 100 height 100
}
```

Listing 8.3: Fine Grained Animation Example

The first will also allow more of the computation to be done at the precompiled system code side of the application, reducing overhead.

8.3.1 Precompiled functionality

Since the scripting solution causes much overhead, it is desirable to reduce the script computation as much as possible and keep as much of the code available as precompiled functions on the system language side. This is important not only to reduce the processing overhead, but also to be able to reduce the power consumption of the mobile device.

We have chosen to use and extend an open scripting platform. These platforms is basically fine grained script languages in the way that they provide support for fine grained logic, and entire programs may be written in them, with only the initiation of the MIDlet and the call to the interpreter in the Java side.

In the first project we developed a model on concepts from different game genres that would work on our project. These can be found in Chapter 12. To provide as much code as possible on the java side, we will have a look on these concepts and new that are under development to make some profiling on what animation elements and communication support that are likely to be needed by the different concepts.

However, we will not constrain the open scripting language further, indeed quite the contrary. We will try to include as much functionality as possible at the script side to allow the game developers to be as creative as possible because we can not foresee every possible scenario. In addition to provide as much functionality on both sides as possible, we will try to keep the size of the classes as small as possible to minimize the storage consumption and heap usage.

8.4 Summary

Grammars for computer languages are concerned by making the language easier to compile or interpret, and to make the syntax and semantics intuitive for the coder to reduce the learning curve and complexity. Our scripting platform for game client development has to inherit the syntax of the open platform we decide to use, and we will therefore take this into account when choosing a scripting middleware platform.

It is also important for a scripting platform running on such a resource constrained device as the mobile phones, to increase the abstraction level as much as possible. Doing so will provide for the client to suffer less overhead runtime, because more of the computation will be executed in precompiled Java. It will also provide for the client development to be easier because of reduced requirements to code needed.

CHAPTER 9

Threats

Pervasive computing comes with a lot of threats that may compromise or constrain the application. Introducing a new layer on top of Java will not make these threats any smaller. In this chapter we will have a look at the technical threats that is relevant for a scripted implementation on a mobile platform.

According to the article *The Challenges of Mobile Computing* [36] the main challenges to mobile computing are Wireless communication, Mobility and Portability. The first two are in our case dealt with by using the Service Frame Framework, with bluetooth communication between the server and the client. Besides, our concept is not mobile in the term of using multiple access points. Portability is an attribute relevant to using scripts with the MOOSES concept.

The attributes mentioned regarding portability issues are battery capacity, small user interface and small storage capacity. This article is written in 1994 and some portability issues has been eliminated or decreased the following years. However, battery capacity remains an even bigger problem today than in 1994 due to that the battery capacity has not been able to increase proportional resources demanded by modern mobile devices. The only way to decrease the power consumption from a software perspective is to decrease the amount of processing needed. Regarding the user interface, modern mobile devices has better resolutions than the previous models. The keypad on the mobile devices may still provide a problem, giving some phones benefits in favour of others, we could solve this by letting the user define his own control setup.

In relation to the storage, our main concern is about the mobile phones memory heap. Mobile phones MIDP stores all application data in non-volatile memory, using a storage system called the Record Management System (RMS)[16]. Modern phones RMS is, in most cases, limited by the size of memory cards that is inserted to the mobile device and may range up to Giga-bytes.

The memory heap however is limited to a few Kbytes on some phones, meaning that we must reduce the need of memory needed by an application to the absolute minimal to expand the number of phones supported.

Using scripts instead of precompiled Java Bytecode provides a span of threats compared to the current implementation of the MOOSES client. These threats are mainly :

Overhead Amount of delay as a result of the additional computation needed to parse and interpret scripts.

Heap Most of the Java supporting phones that are developed today has a memory heap that ranges above one Megabyte, but older or cheaper phones may not. Scripts may increase the need for memory heap.

Battery The Mobile devices has limited battery capacity. The usage of scripts will increase the processing, and therefore consume more power.

Processing A Scripting implementation will cause much redundant processing, because of interpreting and detours of variable references.

It is crucial that the script solution is optimized to reduce these threats as much as possible. To reduce overhead we can start by parsing the script once, in stead of each time it is ran. The scripts may also be parsed on the server before they are sent over, reducing processing needed on the client. Reducing heap size is the worst part. The obvious solution is to make smart solutions for objects that consume less memory. Processing might be reduced by constraining the threads created. It would be possible to make one thread do the work of more threads, since only one thread are executing at a time anyway. For instance, all animations may use the same thread to perform animations.

CHAPTER 10

Script Language discussion

To get around J2ME's lack of user defined class loader, we have decided to base the game applications developed for the client on script technologies. There is a number of scripting languages with J2ME support available, that support an open platform that we may tailor to suit our needs. The prototype will be based on one of these scripting languages, to reduce the development time needed. Some of these scripting languages may prove to fit our concept better than the others with respect to their domain and architecture. We will therefore have a look on the pros and cons for the open scripting platforms aimed for J2ME that we have found and determine which language to go for.

10.1 Simkin

Simkin is a simple interpreted language developed by Simon Whiteside. The language was originally developed using C++ in order to implement an interactive adventure game. The Scripts accessed commands available from the underlying game engine.

In 2000, Simkin was re-implemented to work with Java and XML. The interpreter consist of 100 percent pure java classes. It can used to enable users to customize a Java-based application using scripts embedded within XML documents or TreeNode files (defined at [37]). The MIDP version of Simkin however, does not support ThreeNodes

More information on the Simkin language may be found at the language's homepage [38].

10.1.1 Licensing

Simkin for Java is covered by the GNU Lesser General Public License (LGPL) that can be found at their website [39].

The main points of this license is that you are allowed to copy, modify and distribute the library. Modifications on the library must remain a library. You must notify the changes made, and let the whole of the framework be available free of charge for any third parties.

For all work that uses the library you will have to give notice that the library is used by it and that it is covered by the GNU LGPL license.

10.1.2 Documentation

The language syntax and grammars are well documented on the languages site together with documentation files attached to the downloadable library. The zip file containing the library also includes JavaDoc on the classes in the entire library as well for the Java classes it uses.

10.1.3 Grammar and syntax

Statements and expressions in Simkin is based on infix syntax. This is a strength because it is the most intuitive syntax for programmers, but it might put a constraint on efficiency.

Statements, expressions and procedures are stored and called from an XML file. The complete language grammar may be found at the simkin website.

10.1.4 Size

The size of the JAR file is 87 Kb.

10.1.5 Artefacts

A list of the artefacts in favour of Simkin follows bellow:

- ➢ Powerful language
- ➢ Infix, Left recursive Grammar
- ➢ Well documented

10.1.6 Supported types

Simkin supports the most common types and data structures in software development. Those that is not supported are easily implemented by extending the language. The types supported by default are Integers, Floating point numbers, strings, booleans, characters and objects, which means that any data structure is supported as long as it is defined on the Java side of simkin.

10.1.7 Code Example

```
<example>

<comment>
This is an example Simkin script, it will show some of the
main elements of the language
</comment>

<comment>This field is an member variable of the object owning the script</comment>

<InstanceField attribute="value">InstanceData</InstanceField>

<comment>Here is a more complex member variable</comment>

<InstanceStruct>
<Field1>Field1 Data</Field1>
<Field2>Field2 Data</Field2>
</InstanceStruct>

<comment>Here is a method which takes two parameters</comment>

<function name="Method1" params="a,b">
  return (a+b);
</function>

<comment>This method iterates over the characters in a string looking for the first 'm',
    returning -1 if not found</comment>

```

```
<function name="Method2" params="s">
 length=length(s);      //      call built-in function length()
 index=0;
 while (index lt length){
  if (charAt(s,index)='m'){     //      call another built-in function charAt()
    return index;
  }else{
    index=index+1;
    index=index/1;
    index=index*1;
    index=index%1;
  }
 }
 return -1;
</function>
</example>
```

Listing 10.1: Simkin script example

10.2 Phonescript

Phonescript is a minimalistic programming language based on Postscript [40] designed for Brew and J2ME devices. It emphasize efficiency and is among the smallest frameworks built on J2ME. With the J2ME version weighing in at under 20K Phonescript gives a major advantage over other frameworks.

Additional information and examples of programs may be found at the Phonescript homepage [41].

10.2.1 Licensing

Phonescript is covered under the licensing of TRAC integrated SCM and project management [42] which says:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. - The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

10.2.2 Documentation

Documentation on Phonescript is limited. The languages homepage contains a number of dead links. However, the guide to the script is maintained.

10.2.3 Grammar and syntax

The syntax of Phonescript is postfix. The scripts may be stored in PostScript files and are sent as parameters to the interpreter.

10.2.4 Size

Phonescript comes best out with respect to size, taking only about 20K on the complete framework.

10.2.5 Artefacts

A list of the artefacts in favour of Phonescript follows bellow:

- ➢ Small under 20k.
- ➢ Fast
- ➢ One program can run on J2ME, Symbian, and Brew phones.
- ➢ Can load and run code on the fly.

10.2.6 Supported types

Phonescript support the most common types and data structures used in software development. These are signed numbers, real numbers, strings, arrays, booleans and chars.

10.2.7 Code Example

```
%
% comments
abc%comment(/%) Blah blah blah
%
% ints
123
%
% Strings
(This is a string)
(Strings may contain new lines
and such.)
(Strings may contain special characters*!&|^% and
balanced parentheses ()(and so on).)
(The following is an empty string.)
()
(It has 0 (Zero) length.)
(These \
two strings \
are the same.)
(These two strings are the same.)
%
% Names
abc Offset $$ 23A 13-456 a.b $MyDict @pattern
%
% Literal Names
/abc /
```

```

% immediatly evaluated Names
//abc //

% Arithmatic
12 23  add 2 div .2 pow

%stack manipulation

(first) (2nd) exch
a b c 3 1 roll
a b c 3 -1 roll

%Arrays
[ 12 23  add 2 div
(first) (2nd) exch
a b c 3 1 roll
a b c 3 -1 roll
[ 2 4 sub 4 mul (result) ] ]

%Executable arrays

{ 2 4 sub 4 mul (result) } exec

% Executing strings
(3 2 add) cvx exec

% files
(array.ps) (r) file

% Executing files
(array.ps) (r) file read cvx exec

%control
3 4 lt { (3 is less than 4) 3 4 sub }

0 1 1 4 { add } for
```

Listing 10.2: Phonescript example

10.3 Hecl

The Hecl Programming Language is an high level open source scripting language implemented in Java. The language is small enough to run on J2ME enabled phones. Hecl emphasize being easy to learn and use. Hecl is also intended to be quick to write.

Scripts provided to Hecl emphasis a user friendly scripting interface rather than XML based configurations etc. This means that Hecl parses a plain text file, statement by statement. It is built with a minimal core, supporting most of the logic provided in Java and Object oriented programming.

Additional information on Hecl language may be found at the languages homepage [43].

10.3.1 Licensing

Hecl is available under the liberal Apache 2.0 license [44]. This license indicates that we may use Hecl in our own applications, even if they are not open source, but we have to give the author credit.

10.3.2 Documentation

Hecl is well documented. The Hecl site contains Javadocs for all the classes, brief descriptions on most of the commands, and tutorials and examples on scripts. Also the site contains tutorials on how to interface Hecl and Java, how to manipulate the code and run it.

Hecl has also gained a big community.

10.3.3 Grammar and syntax

Hecl has a prefix notation. A complete grammars of modified Hecl is provided in Appendix B.

10.3.4 Size

The size of the Jar file is 51 K byte.

10.3.5 Artefacts

A list of the artefacts in favour of Hecl follows bellow:

- Easy to manipulate
- Powerful language
- Good GUI support
- Supports fine grained operations

10.3.6 Supported types

Hecl refer to types and objects in form of so called Things (for instance StringThing and IntThing). All types that are supported in J2ME is supported in Hecl. Hecl also support Object oriented programming, referring to the object as an ObjectThing. To interact with objects, Hecl require you to write an interface in form of a Cmds class. For instance the ListCmds class is the part of the interpreter that holds the commands available to work on the ListThing class. The most common objects in Java, such as Lists and Hashtables is included as Things in Hecl.

10.3.7 Code Example

```
#Initiate two numbers
set NumberOne 1
set NumberTwo 2

#procedure that sends a string containing the two numbers to System.out
proc displayTheTwoNumbers {} {
global NumberOne NumberTwo
puts "number one is $NumberOne number two is $NumberTwo"
}

#calling the procedure
displayTheTwoNumbers

#procedure that returns the sum of two numbers
proc addTheTwoNumbers {nbr1 nbr2} {
return [+ $nbr1 $nbr2]
}
#Calling the procedure, storing the return value in a new variable
set NumberThree [addTheTwoNumbers {$NumberOne $NumberTwo]}]

# using the MIDP controls

#procedure that takes the caption from a textfield, reverse it
# and stores it in a static text
proc reverse {tf results} {
# tf = textfield, results = static text
    # get the caption
    set string [getprop $tf text]
    set newstring ""
    # i = length of the text -1
    set i [slen $string]
    incr $i -1
    #reverse the text
    while {> $i -1} {
        set c [sindex $string $i]
        append $newstring $c
        incr $i -1
    }

    #set the caption of the static text
    setprop $results text $newstring
}

# creating a form
set mainform [form label "Reverse a String" code {
        #forms contentpane
    set tf [textfield label "String:"]
    set results [stringitem label "Results:"]

    # adding commands to the softbuttons
    cmd label "Reverse" code [list reverse $tf $results]
    cmd label "Exit" code exit type exit;
}]

# set the form as current screen
setcurrent $mainform
```

Listing 10.3: Hecl example

10.4 Comparison

To compare which language that best suits the MOOSES platform we will divide the script solutions discussed in a table focusing on desired attributes. The results is provided in Table 10.1.

MIDP Scripting Languages Comparison			
	Simkin	Phonescript	Hecl
Documentation	Very good	Limited	OK
Size	87 Kb	about 20 Kb	51 Kb
Licensing	Free and open. The result must be shared.	Free and open	Free and open
Notation	Infix notation	Postfix notation	Prefix notation
Learning curve	Require you to know XML	The Postfix notation is less intuitive than infix	Prefix notation takes some time to get used to, the scripts is in plain text.

Table 10.1: Script Languages Comparison

Phonescript score much points on being the smallest framework, although it may be hard to say how much additional classes and functionality that needs to be added and may expand the size substantial. An inconvenience for this language is that it is hard to find any good documentation on it. To download the library you will have to download one class at a time. We also fear that the small library require more work to be done to the framework itself, yielding a complete library that might range up to the other two in size. The postfix notation that phone Script is based on is a bit harder to relate to than infix and prefix.

Simkin looks like a good alternative. The language has gotten a few years to get rid of the biggest child diseases and is probably the most powerful of the languages we have looked into. However, the XML notation scares us a bit because of the processing overhead XML brings. The framework also contains a great deal of functionality that might be redundant to our implementation. This might be solved by stripping the library of redundant classes when the framework is complete. The GNU LGPL licence says that we have to share the modifications, and that it has to remain a library. The MOOSES implementation is not open. Therefore, such a licence would be inconvenient for us.

As for Hecl, the fact that the scripts are stored in plain text files gives it an advantage in favour of the two others in our opinion. Plain text files are easy to edit without any additional software than those included in every operating system. Hecl is also powerful enough to execute fine grained logic and has good support for Graphical User Interface. It also has an easy to learn modifiability, making coarse grained solutions easy to implement. The licence of Hecl also says that we might use it to projects that are not open source, as long as we give the authors credit.

10.5 Conclusion

All the three languages has their pros and cons. From our point of view, Hecl is the language that covers most of the tasks we have planned for our concept. The first script implementation will therefore be based on Hecl.

CHAPTER 11

Development of the first MOOSES client

This chapter will give an overview of the approach for developing a game client with the former implementation to give an overview of the aim for our scripted implementation.

The mobile client is based on the Java Actorframe framework. This means that it is executed on a finite state machine, which is connected to a server. The state machine handles and redirects messages from and to the server. Graphical user interface is provided to the user in form of the canvas classes provided by Java. The state machine determines which canvas to display, and we may look on the canvases as applications running on top of the state machine.

Development of games to the MOOSES platform needs development in J2ME for the client and C++, Java SE or another language supported by Java native interface (JNI) for the game [1]. The author has to determine the data exchange, and make native interface in the server and the game classes for data exchange. This does not concern the client, as it just receive or sends data in Java format to the server. The data to be exchanged are defined in message classes. The client or server will determine how to use the data based on the message's name. For instance the message UpdateStatusMsg from the first game provides the client with status regarding health, deaths, frags and score.

Before development the authors has to determine which tasks and attributes that are controlled by the client and which are controlled by the game. We can look at the test game to get some concrete examples. We decided that the client should be responsible for keeping track of weapon selection and ammunition status, while the game was responsible for keeping track of the user's health, score, frags and deaths. This means that the game supply the client with concrete status, rather than relative status. Therefore the server is also responsible for knowing

[1]Java native interface is a feature that allows Java to call and to be called by native applications and libraries written in other languages, such as C, C++ and assembly [45].

when a user dies and send a death message to the client.

To direct messages to the canvas running on the state machine, the developer has to manipulate the core state machine, rather than just the canvas and game classes.

11.1 Game client development

The author defines a new class that extends the GuiCanvas class which is part of the ServiceFrame GUI library. This class has methods for user input, drawing and communication. The author writes methods for:

➢ **How the canvas paints** - Every canvas has it's own method for painting. This method is invoked every time the canvas or another class calls the repaint() method.

➢ **Threads for animations or other independent logic** - Things that are independent or concurrent from/to the main thread. Every canvas may start it's own span of threads.

➢ **Receiving and sending data to/from the game** - The developer has to write the routing logic for the state machine, and provide interfaces that the state machine can use to provide data to the canvas.

➢ **How to act on data received from the game** - The developer has to make interfaces that the state machine calls to provide data to the game. He also writes logic for how the canvas act on this data, for instance, update local variables, trigger an event or to display another canvas.

➢ **Reactions to user input** - The developer has to define the logic to be executed when a player presses or releases a key on his mobile. In most cases, a key message is sent to the game, indicating the key and whether it was pressed or released.

➢ **Interactions with other canvases or classes** - A canvas may communicate with another canvas to share data, update data etc. Canvases also have methods for code to execute before it is displayed, and when it is hidden from the display.

In addition to this the programmer of course has to declare appropriate variables to hold the data.

11.2 Development of the SlagMark game client

To illustrate how development might be done at the former MOOSES client implementation we will refer to the game client we developed to the test game during the depth project. The game SlagMark was the game that was used to test the prototype. Every player controls one worm, and the goal is to kill (frag) all the other players. Every worm has an arsenal of weapons he can use to kill the other worms.

In our test game we made inner classes for the threads we used. The only additional class to the main canvas was a canvas that where loaded whenever the user received a KilledMsg. The canvas displayed a picture, name of the player that committed the kill and countdown to the next insertion.

When we discussed how to implement the game concept in the depth-study we had to decide which elements that would be appropriate for displaying on the client display. We found that it would be convenient to display the worm's health on both the client and the main canvas, but to conceal the ammunition status on the client side to make a strategic advantage for the players. We also decided to display the score, frags and deaths on the client to save space on the main game canvas.

11.2.1 Designing

After deciding what elements to represent on the controller, the designing process began, and we found it would be convenient to show the score, deaths and frags in a bin on the top. An icon and a text indicating selected weapon where also put in this bin. Further, we decided to use bar indicators to display health and ammunition status. The layout of the controller can be viewed in Figure 11.1.

11.2.2 Threads

We made four threads to run on top of the canvas. One of them is the reloader thread that runs in the background the entire game session. As the name indicate it reloads weapons whenever they go out of ammunition. The second thread was used to animate the weapon selection, sliding the icon and weapon string vertically, replacing the former picture and string with the new ones. The third thread was used for the rapid fire effect, allowing a user to just hold down the fire button to fire rapidly. The canvas that displayed when the player died used a thread to count down until next insertion.

11.2.3 Communication

We defined four message classes to transfer data between the game and the client, namely UpdateStatusMsg, KilledMsg, ShootweaponMsg and KeyMsg. The first two are used to send data from the game to the client, and the last two messages is used to send data from the client to the game.

In addition to the messages, methods had to be created to send and receive data. The method receiveData, with score, health, deaths and frags as its parameters, is called from the state machine whenever the game updates the client with data updates. To handle the KilledMsg the method receiveDeath, with the killer's nickname as parameter, was created to tell the client that it was killed and had to wait for insertion. This was all the data the client needed from the game. We found that the server would need information about movement and shooting from the client. Whenever a user hit the fire button (or held it down with a rapid fire weapon selected) the shootWeapon method is called with an integer indicating the weapon selected by the user. This method wraps the integer into a ShootweaponMsg and sends it to the server. A KeyMsg is sent whenever the user press or release a movement key, through the sendKey method, with a number indicating which key is pressed and a flag indicating whether it is pressed or released.

11.2.4 Input

Reactions to user input is done by a switch that finds which key is pressed or released and executes the code associated with the key. As mentioned, pushing a movement key (either from the navigation stick or number pad) results in a keymessage sent to the game.Pushing five or navigation stick results in the client sending a ShootWeaponMsg and to execute local code to decrease weapon ammunition etc. The pound sign button triggers the code for changing weapon without sending any data.

11.2.5 Sounds

MOOSES supports up to 21 players, and if all these players should provide actions that generated sounds there would probably be too much noise. With this in mind we developed sound support for the client. Our first testgame was implemented with the gunshot sounds on the client, and the bullet impact sounds on the server.

11.3 Summary

As we have seen, development of a game client to the former MOOSES client implementation is a relatively easy process and it yield good opportunities for visual representation and dataflow. The use of animations and visualization is minimal on the test game compared to potential future game concepts. We plan to make more usage of tactics calculations, extended use of information visualization and even parts of the game play visualized on the client screen. Also, the canvas layout will depend on in-game roles played by the player. This functionality is available now, and it is therefore desirable that a solution for game client development do not put to big limitations on these utilities.

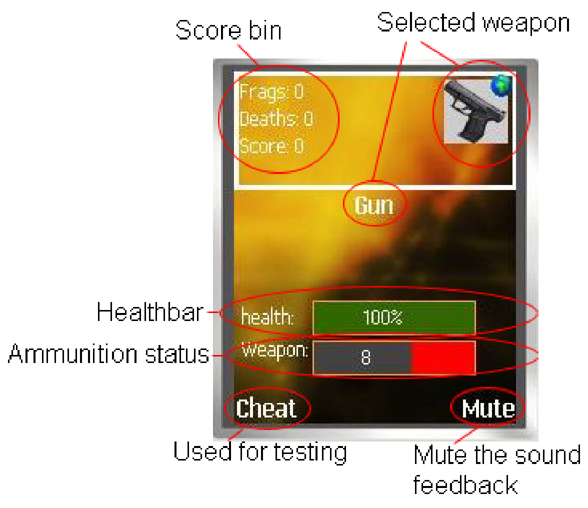


Figure 11.1: Main canvas for the game client

CHAPTER 12

Game concept profiling

To provide for the script solution to our concept to cover most of the most common animations and communication solutions in a coarse grained way, we will look into ways to get it to run on precompiled java side. We will have a look at which animations and communication support that is most common to the given genres or concepts. The discussion will be based on the genres we found appropriate for the MOOSES platform during the prestudy. These genres where, as seen in Table 12.1, Shooter, Third Person (with a fixed camera control), Sporters, Racers, Fighters, Strategies and combination of them.

In this turn we have also extended our focus on games for our concept to include games that emphasize cooperation. In the context of cooperation, new angles of client development are rising. We will therefore present some perspectives of cooperative concepts in Chapter 13.

12.1 Table explanations

Table 12.1 shows our evaluation of different game genres from our depth study. The table is based on the following fields:

Genre - Name of the game genre

Multiplayer opportunities - Indicates to what extent it supports multiplayer for MOOSES and if this multiplayer is based on cooperation or competition or both.

Max Players - The maximum number of players that would be appropriate for the MOOSES for the given genre

Features to concept - Indicates the artefacts that make the genre interesting for MOOSES

Gameplay Elements - Features that is common for the genre

Genre	Multiplayer opportunities	Max players	Features to concept	gameplay elements
Adventurer	Poor, cooperation	2 - 8	Storytelling, enchanting	Solve puzzles, communicate, collect things
Role Playing Game	Poor, both cooperation and competition	2 - 8	Storytelling, statistics, players can manipulate their characters	collecting points and artefacts, develop character, freedom
Shooter	Good, both	About 50	Instant Action, good projection methods	instant action, shoot everything that moves
First Person	Limited, Both competition and cooperation	8 (split-screen)	Same as shooter	Same as shooter
Third Person	Ok, both	About 15	Use of surroundings, isometric battlefield	Bullet dodging, use of environment
Sporters	Good, Both depending on sport	Competitive: 4 - 8, Cooperate: 12 - 22	Endless possibilities, every sport can be used	Depending on sport, should be implementable to our concept
Racers	Good, competition	4 - 16	Either visualize from top, perspective or split-screen	Competition and instant action
Fighters	Good, both	4 - 16	Cooperation in beat em' ups or Teamed competition in versus fighters	Instant action, fighting opponents
Simulations	Difficult, cooperation	Depends on simulation	Maybe in an extension	Often Educational, demands patience
Strategies	Good, Teamed competition	2 - 10	Reduced version, team based, limited compared to the originals	Time demanding, collect resources, build army, make strategic moves

Table 12.1: Genre sufficiency from our depth study

12.2 Shooter profile

A Shooter (often referred to as shoot em' up), is a computer and video game genre where the player has limited control of their character or machine. The gameplay focus is almost entirely on annihilation of opponents or objects by aiming and shooting projectiles in their direction [46]. An overview of some of the most common shooter sub-genres can be found at wikipedia [47].

The first game we made to the MOOSES platform was a shooter, and therefore we have some experience on which attributes that would suit the genre. The shooter is probably the genre that allows most freedom for creativity and that can be merged with other genres. Our first game was a strategy shooter, giving the player time to aim and calculate where his bullet would hit.

12.2.1 Feedback

In a shooter the players often has an arsenal of different weapons. It would be convenient to display for the player which weapon he has selected on the client display, not only to save space on the big screen, but also to give a strategic advantage of hiding this information. Health is also an important feedback for a shooter player. Close to all shooters have some kind of health barometer to provide feedback to the player about his health condition. Whether or not this information should be hidden for other player may be up to the developers.

Other attributes that is common in shooters is amount of kills (frags), score, deaths, lives left, ammunition status. Other statistics such as accuracy, highscore position and time alive etc. is usually provided at the end of a match or level.

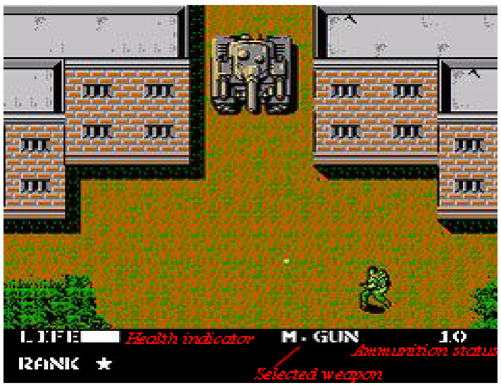
12.2.2 Visualization

The feedback mentioned above may be visualized only by a text indicating the value or by barometers that indicate the value compared to the maximal value. Such a bar may be animated or just calculated by the value it is indicating using graphical primitives. Animation of a bar will be visualization of it decreasing or increasing whenever it is updated with a new value. Typically if it is a black color indicating the true value, a red color for the background, you would have another color indicating the interval decreasing or increasing and cut this down with a timer until it reaches the actual value.

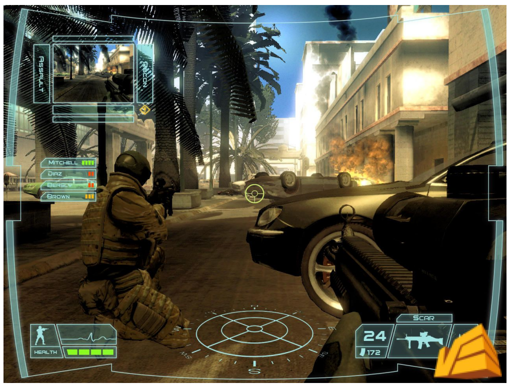
In our test game implementation we used a image and string rotator to display the current weapon selected. The first game client we tried to implement when we first tried the Hecl framework was the client to the MOOSES game. When i wrote the code for the Image rotator (after a few modifications of the Hecl source) we experienced a huge overhead that made the rotator useless. Some modifications reduced this overhead, but it would be convenient anyway to have these rotator elements implemented as compiled java code to reduce overhead and processing for saving battery power.

12.2.3 Screenshots from shooter-games

We will provide some screenshots in Figure 12.1 to illustrate common solutions for feedback and visualization for shooters. Metal gear is an old stealth shooter. As displayed in the screenshot, the player is given feedback on health, ammunition and selected weapon. The player may change weapons and equipment through another interface, displayed on demand of the player. Metal gear has a pretty simple gameplay in contrast to Ghost recon advanced warfighter. As the screenshot shows, much more feedback are given to the player in ghost



(a) Metal Gear



(b) Ghost recon advanced warfighter

Figure 12.1: Screenshots from different shooter games

recon. First person shooters will not be relevant to our concept (at least not on the main game canvas), but some of the feedback mirrors feedback we would like to provide in our game. The player is given feedback on selected weapon, both by displaying it and by text and icon in the lower right of the screen. He is also given feedback on ammunition status, both on the weapon and ammunition left. A compass shows the orientation of the player. He is also given status of his team-mates and as we can see in the upper left corner, he is given the view of one of his team-mates. Health and stance is provided in the lower left corner.

12.3 Third person profiling

Third person games is a genre that display the character or characters controlled by the player from a third person perspective. They emphasize usage of surroundings, for instance use of obstacles for cover etcetera. Third person games often involves control of the character and the camera. The gameplay on third person perspective games will change on our concept compared to what's common on single terminal games. We will have a battlefield or world viewed from a fixed perspective angle, where players may use the environment tactically to proceed. When talking about third person games, many refer to these games just as third person shooter (TPS). This is because the shooters are dominating the third person genres, and the gameplay on other third person action games are pretty similar to the TPS genre. However, third person is just a perspective that may be used for almost all genres such as sporters, racers, strategy, adventure etc.

Although we have not made a third person game yet, we have some good ideas on how to do it. Another project that runs in parallel to this one focus on development of new game concepts for MOOSES, and one of the test implementations is a third person shooter with fixed camera control. The game will be a space shooter viewed from an automatically zooming perspective angle. All players may choose or gain roles as they play, and their roles will affect their gameplay and the team's strengths. More info on this game can be found at [5].

12.3.1 Feedback

The feedback attributes on third person games are common to the one's for shooters, although there are some additional.

Players may need feedback on their position, location of targets or strategic points, or battle strategies. This is necessary because of the new game concepts that require more communication between players. A brief overview of the new ideas to new concepts is given in Chapter 13.

The feedback given to the player is also dependent on the role the player has chosen or been given. Although third person is commonly associated with shooters it might be fitted onto other genres, making their feedback of relevance.

12.3.2 Visualization

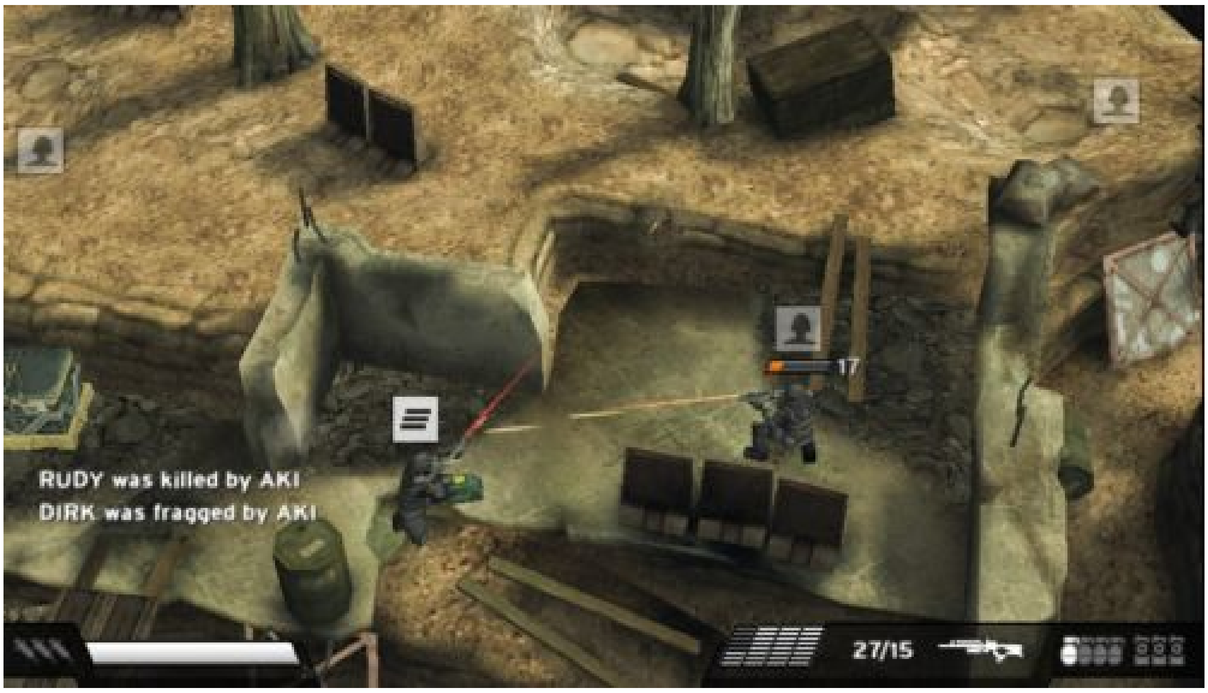
Feedback on players' position and interesting points on the map such as targets, vehicles, strategic plans, cover spots or other objectives may be provided to the player through a minimap. A minimap would probably not require animation, but support for visualizing spots in form of small images or dots. It would also need support for drawing routes to certain points. Animation elements required would be the opportunity to make icons or dots blink to get attention. A more advanced minimap could support zooming and more detailed animation of movement or changes.

12.3.3 Screenshots from Third person games

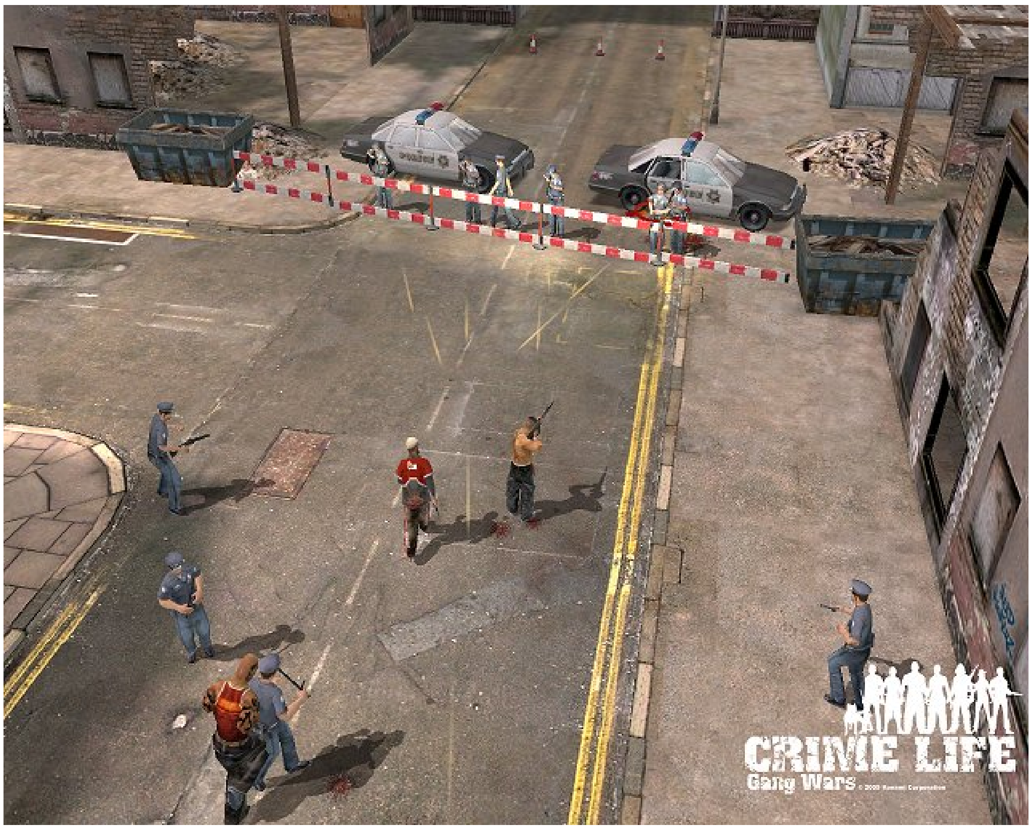
Common solutions for feedback and visualization from various third person games is provided in Figure 12.2. The two games both represents games that we want to gather inspiration from in further development of new games. Killzone is a shooter with multiple players competing on a battlefield that allows them to take advantage of environment. For instance to seek cover behind different obstacles, to shoot barrels of explosives near an enemy or to use mounted weapons or vehicles that are located around the battlefield.

As displayed in the screenshot, players are given feedback on their selected weapon, ammunition and grenades left and health status. Players' stats are displayed on the side, and the player's position is provided by keeping focus on it in the center of the view.

The Crime life screenshot does not provide much feedback to the player. Crime life has a simple gameplay that only need to display the condition of the player. However, it shows potential usage of a fixed third person angle to display a greater battlefield with more characters, which is the goal of our next game.



(a) Killzone: Liberation



(b) Crime life, Gang wars

Figure 12.2: Screenshots from different third person games

12.4 Sport Games profiling

The Sport games genre span over a large number of games tied to most of the sports represented in the real world. The gameplay on these games are typically to give a player control one or more of the characters in setting of the sport.

Sport games represent a problem for our concept, especially regarding cooperative games. Our concept needs the availability for players to come and go from a running game as they plead. Also, if the players should only control one player, it would be inconvenient for few players in team based sports. Therefore, it would be a suitable solution to provide control of a bundle of characters from a fixed number balanced on the amount of players participating.

As mentioned in our depth study [7] we found sporters suitable for the single screen multiplayer system. The best support for such games would be games that are based on team sports. Such as Ice hockey, soccer, football, basket etcetera.

12.4.1 Feedback

Given that players should be able to come and go as they please, the player may be forced to change team during the game session to balance number of players on the teams. It is therefore important to give the player feedback on which team he belongs to.

Team based sport games usually provide scores both for the team as a whole and for the player's achievements. Depending on sports, a reflection on the achievements would be appropriate, for instance status on the game such as the goals or points and which team is leading.

The different characters may have different skills or purposes. Interface for showing or using these may be of relevance. A player that has based his character on speed will be able to outrun many of the opponents, but might lack ability to hold control of the ball for longer periods or shoot less accurate.

12.4.2 Visualization

As mentioned, a player needs feedback on which team he belongs to. This could be done by showing the team's icon in the display with a topic indicating the team. Also, the environments in the display, for instance background color, could be filled with the team's colours.

Scores and achievements may be displayed using a progress bar or just string representation of the values. For example, we could use a bar to balance which team is in the lead and the strength of the lead.

Players may be able to balance the skills of their character(s) by balancing points into groups (stamina, strength, speed etc). A common way to implement this is to provide a list with the different attributes with a fixed amount of points to add to desirable categories.

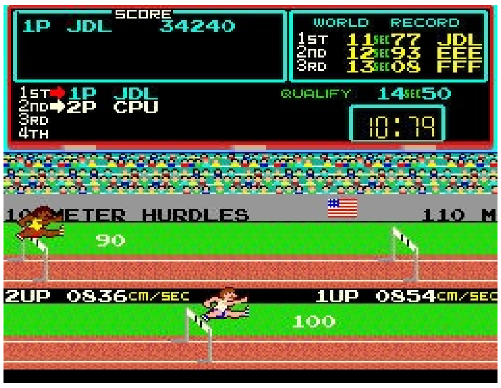
In many sports different roles of a team has slightly different gameplay. For instance, baseball, which has a thrower, a batter and people on the field to catch the ball. Visualization of gameplay attributes on these roles will be different. A thrower will have visualization on aim, and how to throw (direct fast, screw or bow), while a batter will need to balance something to hit, and bars to indicate strength etc.

Although sport games for MOOSES are likely to display the entire map on the projected screen, the client could take advantage of having a minimap for the games that focus on scrolling, showing where the camera is displaying in relation to the entire map.

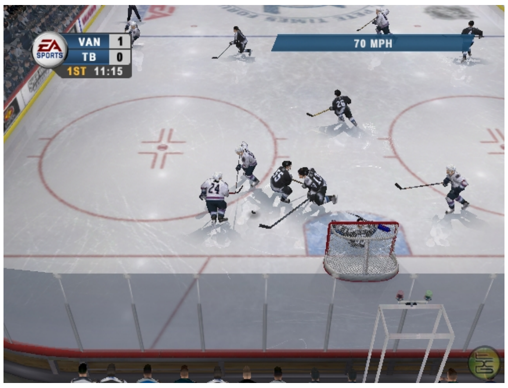
All in all, team based sporters may need support for the minimap, animated bars, and lists.

12.4.3 Screenshots from sport games

Common solutions for feedback and visualization from various sport are displayed in Figure 12.3. The first game, Track and field, is an old athletics game from Konami with a rather simple gameplay. The player is given feedback on his position and the time elapsed.



(a) Track and field



(b) NHL 06

Figure 12.3: Screenshots from different sport games

NHL gives the player feedback on the team's goals, which team is leading, and which character the player currently controls.

12.5 Racers profile

A racing game is any game that involves competing in races through a surrogate playing piece or vehicle, either getting it from one point to another or completing a number of circuits in the shortest time.

Racing games might be hard to implement on our concept. However, there are some alternatives that would allow for it to be done. One problem is, as with sporters, it will be hard to allow for users to come and go as they please. Because a race usually take place between two points, or during a fixed number of laps, it would be difficult to shoot in a new player in the middle of it. One solution to this could be to provide a couple of artificial intelligence (AI) controlled cars that could be taken over by players and let AI take control of cars whenever players leave. Another problem is the visualization because as the distance between the vehicles increases it is harder to display them in one screen. To be safe, the best way of making a racer to our concept is probably to display the whole course at the canvas, or forcing the players to stay in scope to still be in the contest.

12.5.1 Feedback

The typical user feedback from racers are speed, warning of upcoming turns or obstacles, and car condition.

As with most of the other genres, racers may provide some custom additional attributes to let the player customize the car or character.

For games that do not display the whole course on the the main screen, it would be convenient to display the players position compared to the other players and the course.

12.5.2 Visualization

The most intuitive way to reflect the players speed would be a speedometer. This should provide animation opportunities with a image as background, and an image or line as pointer. This animation element could be used by other game concepts to indicate intervals. For instance strength in baseball strikes etc.

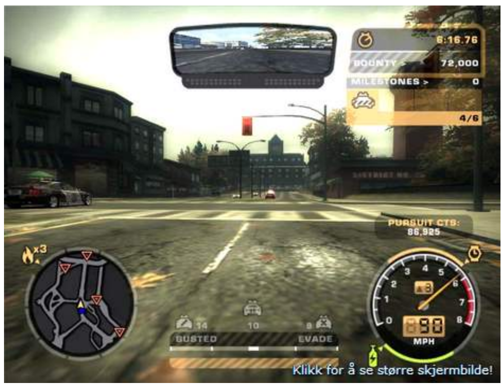
To display the players position in relation to the track, many racers have used a vertical pole that has the start position at the bottom, and the goal at the top. The players current position is indicated by a line or dot at this pole, giving the player information on how much he has left behind and how much is left of the track. The pole may also contain checkpoints and so on to give a better feeling with it.

Another way to reflect the players position could be to use a mini map that draws the route, and have different colors on the part of the track left behind, and the part of the track remaining.

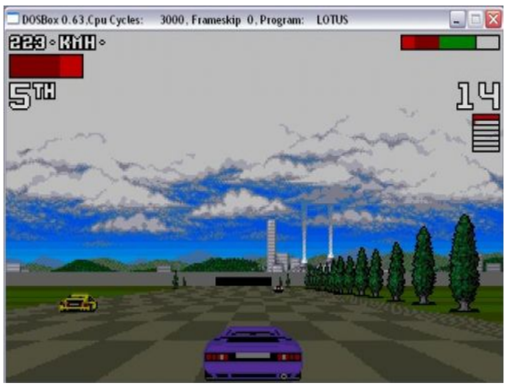
Racers usually provides for the user to choose the vehicle he shall use during the race. Interface for vehicle selection could be provided either through lists, or by use of a canvas that displayed multiple icons, where each icon served as a button. The user could navigate to the image of his choice, and confirm selection by hiting the fire button. Alternatively an imagerotator may be used to select vehicles.

12.5.3 Screenshots from Race-games

Common solutions for feedback and visualization from various race games are displayed in Figure 12.4. Need for speed has gone into the history as one of the most popular car game series. As we can see on the screenshot, the player is given feedback on his speed, the mirror gives a view backvards, minimap providing position and an indicator at under the speedometer gives the amount of nitro left.



(a) Need for speed most wanted



(b) Lotus III

Figure 12.4: Screenshots from different shoter games

Lotus III is a third person racer. In the upper left corner we se feedback on players position in the race, speed and rational speed. The player is also given feedback on lap-progression and laps left.

12.6 Strategies profile

The strategy genre is usually set to command one part of a battle. The gameplay is either realtime or turn based. The traditional strategies developed for consoles and computers have a scrollable view over the battlefield, and provides for the user to use the mouse to control units on the map, and quick-buttons on the keyboard to do frequent occurring tasks. The player control a collection of units that he use to defeat the opponent. Stragies often involve reource management to acheive new units to control.

It is obvious that the interface has to be changed to fit our concept, not only for the controller but also the visualization techniques since we only have one screen to display all the action. Our implementation would probably require just to command the units directions and attack options.

12.6.1 Feedback

Strategies is probably the genre that depend most on the opportunity to conceal data using the client. Much of the info that is required to be feedbacked to the user is therefore preferably displayed on the client.

Much of the feedback that is needed to be provided to the user is already displayed on the main canvas. Commonly in strategies, a player will be notified whenever one of his units is under attack. This would probably be redundant in our case, since the whole map is displayed.

A common feature in strategies is resource management, both available units and resources collected. The player would need feedback on resources, and interface to manage them.

The player would also need feedback on his units condition and attack adjustments.

If the game supports structure building and/or ordering of new units, the player will need feedback on which buildings and units that are available.

Another feedback that is required to be reflected to the user is the strategies available in given situations. For instance for attack, defense and management.

12.6.2 Visualization

All the users and the opponents units are displayed on the main canvas, and therefore reflection on which units that are attacked may be redundant. However, it might be appropriate to fetch the players attention. This could be done by using the vibrating in the mobile phone, or blink the lights or an icon.

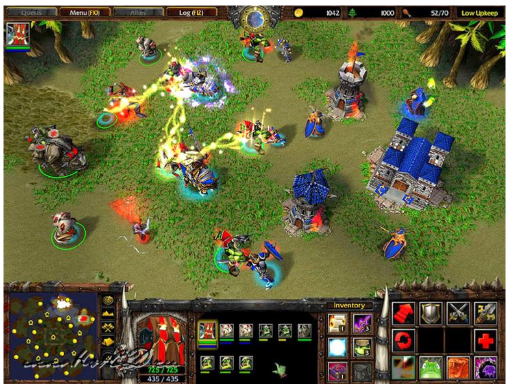
For resources to collect, the most common solution is to provide this as a string value indicating the amount collected. Reflection on unit management is another story. It could be appropriate to display the units in a hierarchy list, depending on the amount of units available to the player. The player could select a unit from its subgroup, and his repetoir of that unit would come up in a list showing their condition. A further click in the list would result in control of the particular unit, for instance a display with a picture of the unit and the options attached to it. A similar list could be used to order new units.

To build structures, a player could view the structures available through another list. The best solution to deploy the structures would be to use fixed locations on the main canvas. Alternatively, the user could navigate to the desired deployment location either through a minimap or a cursor at the main canvas.

To interface with strategies it would be convenient to provide these in a list or a buttoncanvas, as shortcuts.

12.6.3 Screenshots from strategy games

Common solutions for feedback and visualization from various strategy games are displayed in Figure 12.5. Warcraft III is a popular realtime strategy games that includes building bases, and management of resources and warriors. The player is given feedback on available resources at the bar in the upper right corner. In the lower left corner we see a minimap. This minimap gives a concentrated view of the whole map, and highlights important loacations. If one of the players warriors are attacked, it will also be higlighted on the mini map. The panel in the lower right corner provides strategic options for the player. This panel shows the idea of a buttoncanvas to interface with strategies.


(a) Warcraft III

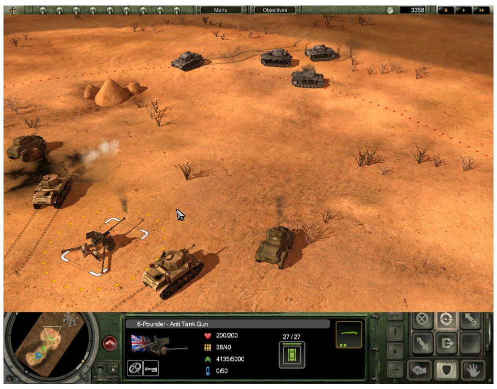

(b) Codename panzers

Figure 12.5: Screenshots from different strategy games

Codename panzers is a realtime strategy game that emphasize the battle and leaves the resource management to before the battle begins. The feedback layout is basicly like warcraft. Minimap in the lower left corner, and strategic options on the lower right corner. In the middle of these is a panel that shows the status of the selected unit(s), such as ammunition status, condition, crew etc.

12.7 Fighters profile

There are two kinds of fighters, versus fighters where players compete each other face to face, and beat 'em ups where one or more players fight a horde of AI controlled enemies while travelling through horizontal scrolling levels.

The player typically controls one fighter, and resort to tricks by dialling different key-combinations. A versus fighter typically hosts for two fighters at the time, and beat 'em ups typically support two to four for multiplayer.

12.7.1 Feedback

The player needs feedback on his position in relation to the others. He will also need to know which characters that are available for character selection, and eventually the key-combinations available for the selected character.

As for beat 'em ups, the player will need reflection on number of lives left and maybe progression in the current level.

12.7.2 visualization

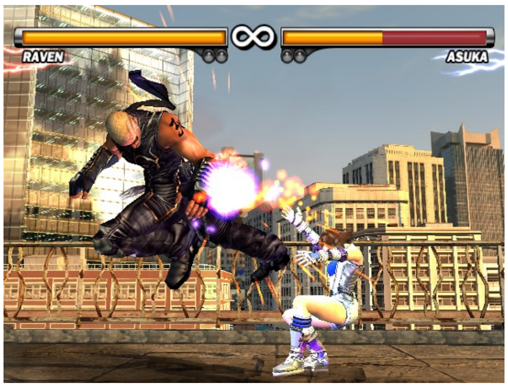
Fighters may have a relatively long learning curve based on how advanced they are. There might be about hundred different key-combinations tied to every playable character. Common ways to provide for the player to learn these combinations is to include them in the games manual or to make tutorials where the player can learn to use the characters. To ease the learning curve we would probably have to cut down on the number of key-combinations to between five and ten. Some combinations could be common to all characters, while a few special is tied to the different characters. The combinations would probably have to be stored in a list on the client, which the player could view while playing. The tutorial part would be difficult to implement on MOOSES, because it would be inconvenient to lock the main canvas to provide for a few players to practice. Making the tutorial local on the client is an option, but it would not provide the right feeling with the main game to the player. The best way is probably to have a vote during the start of the game on whether or not the first two to four minutes should be used to practice.

The player's condition is commonly provided by giving all of the fighters' health-bars showing how much more they can take before they fall. Whether or not this should be provided on the main canvas, both the main canvas and the client, or concealed by the client is up to the game developer.

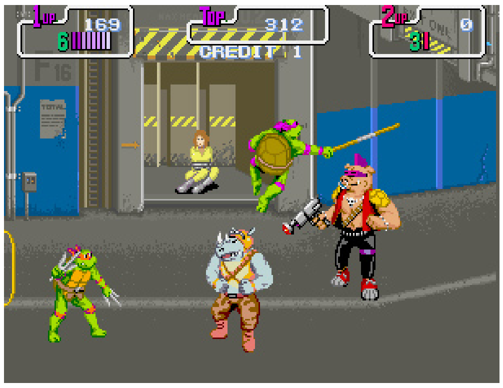
Just as the racers, fighters usually include character selection before the game takes place. Available characters could be displayed in a button canvas or using image rotators.

12.7.3 Screenshots from fighter-games

Common solutions for feedback and visualization from various fighters are displayed in Figure 12.6. Tekken is a fighter series that in the newer versions supports both versus fighting and beat em' up mode, although it is commonly known as a versus fighter. In the screenshot we can see that players are given feedback on their health. The circles under the health bar indicates rounds won. Between the health bars is a field used for round time countdown, which in this case is turned off. Tekken also support an interface for showing button combinations on the screen, making it easier for the player to familiarize with them.



(a) Tekken 5, versus fighter



(b) Ninja Turtles, beat em' up

Figure 12.6: Screenshots from different fighter games

Turtles is a beat em' up game. In this game the players are given feedback on their health, lives left, credits and score.

12.8 Profile Summarize

The profiles shows that much of the information that needs to be reflected to the player may be displayed just as text representation or icons, with icons being preferred because it is easier or faster to interpret by the player and contribute to the game atmosphere. Some of the feedback is best given by graphical objects and animations of these.

The feedback and visualization methods for the different genres are displayed in Table 12.2.

12.8.1 Graphical elements

For a scripted solution to the MOOSES client, all the graphical elements has to be ready on the Java side and then glued together with the script. Some of these elements will be ready to the prototype developed in this run, and some will have to wait. The elements that have priority

Genre	Feedback	Elements needed
Shooter	Weapons, ammunition, health, frags, kills, deaths, lives left	Text, icons, progress bars, image- and stringrotator
Third person	Same as shooter, position, role specific information, team mates, strategic points	Text, icons, minimap, progress bars, rotators
Sport games	Team indicator, stamina, game status, skills	Text, icons, bars, lists, minimap
Racers	Speed, upcoming turns/obstacles, car condition, car customization, position, available cars	Text, icons, mini map, speedometer, progress bar, pole, button canvas, imagerotator
Strategies	Location of units, unit status, resources, available upgrades, available strategies	Text, icons, mini map, lists, button canvas
Fighters	Health, round time, progression, available characters, key combinations	Text, icons, progress bars, list, button canvas

Table 12.2: Elements needed by the different game genres

are the elements we used on our previous implementation. These elements are prioritized to be able to test the scripted solution with the game we have already developed, at Kosmorama film festival. The elements that will be developed for the prototype are:

Rotators The Imagerotator and String rotator are elements that can be used by most games developed for MOOSES. In our first game, SlagMark, we used a synchronized stringrotator and imagerotator to display the arsenal of weapons available. The scripted client will primarily be tested with the SlagMark game.

Animated bars These bars are easy to build from scratch, since they comprise a few graphical primitives. However, providing support for them on the java side might reduce the script latency.

Icons Icons might be implemented as animations as well, a picture blinking for instance. The prototype will support static Icons in the first turn.

Text Is supported by J2ME, the script just need an interface to use it.

The other elements are not crucial for any of the games we have developed so far. Some of them, such as the mini map require a lot of planning for foreseeing a solution of implementation. The minimap would probably have to be processed mainly on the server. Maybe split up to cells to decrease the amount of the map needing processing. These are the elements that will be implemented at a later stage:

Speedometer - Implemented like a progress bar. May be used in several games, but mainly car games. Convenient to use an image as background.

Lists - The client might take advantage of using the a modified version of the lists provided by the service frame GUI library. However, lists will also be implemented to the scripting platform later.

Button Canvas - Button canvas is intended to be a selection of icons where the user selects one icon at the time, navigating either horizontal or vertical between the icons. Used for character selection etc.

Mini map - A concentrated map of the main game canvas. Mini map can support important locations in form of small dots, and drawing of paths between them etc. They may be used by several games.

Pole - A pole with a starting point and an endpoint to display checkpoints and progression.

12.8.2 Game development support in MIDP 2.0

In addition to these graphical elements and animations we might take advantage of the MIDP 2.0's support for game development. The game development framework provided by MIDP 2.0 comprise functionality for character animations and collision detection using Layers [48]. This framework raise the abstraction level on game development for mobile phones. If we could interface with it from the script, we might be able to provide a platform for creating more advanced gameplay on the client. The framework could also ease the implementation of a mini map, using Tiled layers which divide images into regions. Using Tiled layers we would only have to update the cells of relevance.

Implementation of the MIDP 2.0 game development support would require some major changes to the client. We will look on this opportunity if the scripted implementation proves to be a success.

CHAPTER 13

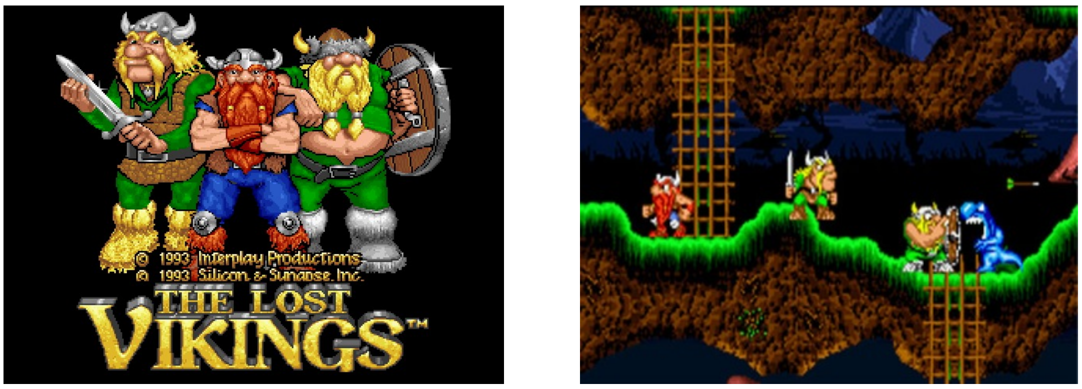
Cooperative Games

Together with the profiles tied to the genres, there are other aims that has impact one elements that need to be covered in a scripted client. During the first project we neglected the support for direct communication and data exchange between the players. This was done with respect to the time available for development of the prototype, together with the fact that since the test game was based on competition between all the participating players there would be no particular need for direct communication and data exchange. However, the idea of cooperative games seems to suit the concept even better than plain competitive games. Therefore, we will have a look on solutions to implement direct communication between participating players, and how to implement this to a development platform.

In our prestudy on the MOOSES framework [7] we discussed different aims for multiplayer. Namely cooperation, competition and socialization, with the last one commonly achieved by the two first. We discussed the opportunity to make use of cooperation to our concept, and concluded that it might be a good idea to divide the players into two or more competitive teams.

To draw this further, we may also take cooperation to an extent where it forces the players to cooperate. In most cooperation based competitive games the players are free to act on their own and does not really rely on the team-mates. By giving the players unique skills that is needed for the task to complete the teams objective or beat the opponents, we could assure cooperation between players on the same team.

To draw an example picture of these skills we can begin by referring to the old game Lost Vikings. Screenshots from the super Nintendo version is provided in Figure 13.1 The game is single player, where the player controls three Vikings that are sent through time. One Viking can run faster than the other two, he can jump, and can bash through some walls (and even some enemies) with his helmet. The second Viking can kill enemies with his sword, or from a distance with his bow. The bow can also be used to hit switches from a distance. The third Viking can block enemies and their projectiles with his shield, use his shield as a hang glider, or as a stepping stone for the first one to enable him to reach high grounds which is not possible without the shield. The whole point is that you need all the three Vikings abilities to complete the level. More information on Lost Vikings may be found at wikipedia [49], or at the blizzard homepage [50] which provide a flash demo of the game.

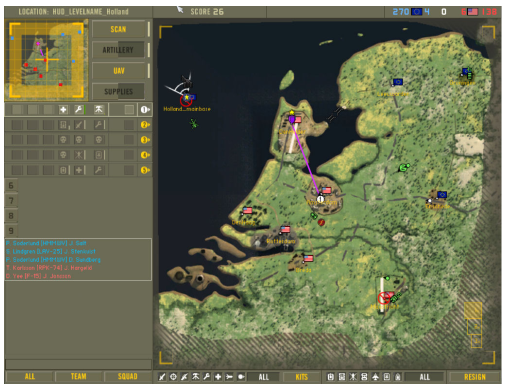


(a) Lost Vikings title, Super Nintendo version (b) Lost Vikings in-game screenshot, Super Nintendo version

Figure 13.1: Screenshots from Lost Vikings

The cooperation based on dependence on the other players' roles will probably enhance the gameplay and socialization experience, but it will constrain the concept in the way that it depends on a lower threshold of players to provide a good gameplay. If there is not enough players to fill the required roles, players might not be able to reach certain points or complete objectives. This could be solved by making the environments aware of which roles that are taken and adjust obstacles and so on to fit number of roles taken. But other utilities may be satisfied to emphasize cooperation.

When talking about team based cooperation the wave of first person shooters with team based multiplayer is a good example for providing team based gameplay. This genre was grew popularity with the Counter Strike mod developed for valve's Halflife game. Counter Strike divides players into two teams, where one of the teams serve as terrorists that either kidnaps hostages, try to blow of a bomb or kill a VIP. The other team, Counter terrorists, is a police group that tries to stop the terrorist by killing them [51]. The idea of this game is to get the members in the teams to cooperate by combine tactics and fight together to solve the objective, but there is nothing that directly force cooperation between the players. A player is free to roam alone as he please, and during online sessions with players that does not know each other this is usually what happens. Therefore, games based on the same concept that has been developed in the later has developed new ways to achieve team-play. One great solution we can take advantage of is the technique used in Dice's Battlefield 2. Battlefield has a wider aim than counter strike with bigger maps, support for more players (up to 64 [52]), use of vehicles and air force and roles for each player such as medic, support gunner, engineer, assaulter etc. As for the team, the players joins one of the squads on the team. The squad gains points by cooperating, which ties the players more tightly to cooperation. Each squad has a team leader that commands tactics on a map, and can ask for support such as resources or artillery. In addition to this each team has a commander that commands all the squads to attack an area, defend an area etc. The commander also respond to queries from the team leaders. Screenshots from Battlefield 2 are provided in Figure 13.2.



(a) Battlefield 2, Commander view

(b) Battlefield 2

Figure 13.2: Screenshots from Battlefield 2

A possible cooperative gameplay discussed for our concept is to make a battlefield viewed from an angled perspective, with two bases with a corresponding team. The objective is to take over the opponents' base. By giving the players a spectre of different roles to chose from, for instance field Engineer and demolition solider, which have different tasks relied upon them to get to the opponents base, for instance blow a way through obstacles or repair vehicles etc. And for controlling vehicles, different roles are required. As one controls the vehicle, other players may function as gunners etc.

The gameplay descriped above requires direct comminication between the users. This could be achieved in multiple ways. User input in form of text would be little convenient with respect to time required for writing them. However solutions with predefined words could be sent. Also, team-mates could be synchronized in given situations, for instance by sitting in the same vehicle or just by teaming up. Relevant data could be sent between the players, and developers could take of this utility.

Since the communication between the server and the clients is based on bluetooth, it would be desirable to use a bluetooth peer-to-peer solution without having to go through the server for direct communication between the players. Unfortunately this has proven to be difficult due to the Bluetooth lack of scatter net support [53].

This means that most of the work that has to be done to the framework making it able to support direct communication between players lies on the server side of MOOSES and not the client. The server has to provide lists of available users, and maybe information capsules in relation to game data, allowing two or more users to team up and communicate directly. We will also need opportunities to interfere with this information from the script.

CHAPTER 14

Hecl - The mobile scripting language

In Section 10.5 we decided to use Hecl as a base for the scripted client prototype. In this chapter we will have a deeper look on Hecl and it's architecture.

The Hecl Programming Language is a high-level, open source scripting language implemented in Java. It is intended to be small, extensible, extremely flexible, and easy to learn and use. Hecl is small enough to run on J2ME-enabled cell phones [43].

Hecl is intended as a complement to Java, and does not aim to replace it. This means that it tries to do well what Java does not, and this is where it fits into the MOOSES project, providing a substitute for the user defined class loader that is not present in the current version of J2ME's CLDC. By authoring the MOOSES game controller application through scripts, the client would only have to download the script together with it's resources to get the controller application for a certain game on MOOSES. The resources can be put into the Mobile Phone's RMS and accessed dynamically runtime by the MOOSES client application.

14.1 Hecl Architecture

The Hecl scripts give opportunities to declare variables represented as text strings, which is put in a hash-table on the Java side and is referred to at the Java-side as so called Things. A Thing is a Hecl object translated to Java. A Thing is an encapsulation of a RealThing instance, which is a Java object in Hecl.

When executing a script in Hecl, hecl first parses the script by sending it through Hecl's lexical analyzer. The lexical analyzer returns the parse tree, referred to in Hecl as a collection of stanzas, that are sent to the interpreter that executes the stanzas.

14.1.1 Example: Adding a new Thing to Hecl

We can make a new Thing-class to hold Image values in Hecl. Normally this could be done by using the ObjectThing class, but we may also make a new class that implements the RealThing interface. In this class we can provide methods for operating on the object. In our example, we have a method in addition to the constructor that provides resize functionality for the image. Listing 14.1 shows the source code of the ImageThing.

```java
public class ImageThing implements RealThing {
        Image val = null;
        String path = "";
        int width = 0;
        int height = 0;


        /**
         * Constructor - creates a ImageThing instance and initiates the Image
         * */
        public ImageThing(String Path, int Width, int Height){
                val = Image.createImage(Path, width, height);
                path = Path;
                width = Width;
                height = Height;
        }

        /**
         * Resize's the Image
         * */
        public void setSize(int width, int height){
                val = CanvasUtil.resize(val, width, height);
        }

        /**
     * The <code>deepcopy</code> method must copy a RealThing and any values
     * it contains.
     *
     * @return a <code>RealThing</code> value
     * @throws HeclException
     */
    public RealThing deepcopy() throws HeclException {
        return new ImageThing(path, width, height);
    }

    /**
     * The <code>getStringRep</code> method returns the string representation
     * of a <code>RealThing</code>.
     *
     * @return a <code>String</code> representation of the value
     */
    public String getStringRep(){
        return ("ImageThing path: " + path + " width: " + Integer.toString(width) + " height:
            " + Integer.toString(height));
    }
}
```

Listing 14.1: ImageThing class

14.1.2 Hecl Things

In this section we will present the things available in the Hecl core by default.

CodeThing - Holds a compiled code segment that might be tied to an instance of an object

DoubleThing - Holds a Double value

FractionalThing - Is a super class for floating point variables

GroupThing - Is used by the interpreter to keep components together

HashThing - Represents a hashtable type in hecl

IntegralThing - Represents integers or longs

IntThing - Represents an integer value

ListThing - The ListThing class implements lists, storing them internally as a Java-Vector

LongThing - Represents a long value

NumberThing - Is the super class for all the Things related to numbers. It contains static methods for calculations.

ObjectThing - Represents a wrapper for objects that are not representable as Strings. May be any object in Java.

PrintThing - Is a utility class used to print out Things. It is useful for debugging purposes.

RealThing - RealThing is an interface which all things must implement. All Things has an attribute *val* that represent the object or type which is the value of the Thing.

StringThing - Is the internal representation of string types. This is somewhat special, as all types in Hecl may be represented as strings. All things are initiated as StringThings (With the exception of objects) in Hecl before they are casted into other things on the Java side.

SubstThing - Represents a thing that must be substituted, for instance $hello or &hello. This class is used by the parser

Thing - Is Hecl's representation of objects. All RealThings can be encapsulated by a Thing representing it.

These Things are part of the Hecl core, and new things might be added as an extension. The logic in hecl is evaluated by the interpreter through a collection of command classes. Command classes are also written to provide commands to operate on Objects or Things.

14.1.3 Command class Example

As an example, we can make a Cmds class to work on our ImageThing class. Listing 14.2 shows the Cmds class for ImageThing.

```java
package org.hecl;

public class ImageCmds extends Operator implements HeclModule {

        // Constants
        public static final int IMAGE = 0;
        public static final int RESIZE = 1;


        /**
         * Constructor - makes a new command avalable
         *
         * */
        protected ImgCmds(int cmdcode, int minargs, int maxargs) {
                super(cmdcode, minargs, maxargs);
                // TODO Auto-generated constructor stub
        }


        /**
         *
         * Operate is called by the interpreter whenever one of the commands held by the
         * ImageCmds is called.
         * */
        public RealThing operate(int cmdcode, Interp interp, Thing[] argv)
                        throws HeclException {
                // TODO Auto-generated method stub
                switch(cmdcode){

                // Create a new ImageThing
                case IMAGE:
                        return new ImageThing(argv[1].toString(),
                                        IntThing.get(argv[2]),
                                        IntThing.get(argv[3]));

                        // resize a imageThing
                case RESIZE:
                        ((ImageThing)argv[2].getVal()).resize(
                                        IntThing.get(argv[2]),
                                        IntThing.get(argv[3]));
                        break;

                }
                return null;
        }


        /**
         * Interface to load the commands to the interpreter
         *
         * */
        public void loadModule(Interp interp) throws HeclException {
                // TODO Auto-generated method stub
                super.load(interp);
        }

        /**
         * Interface to unload the commands from the interpreter
         * */
        public void unloadModule(Interp interp) throws HeclException {
                // TODO Auto-generated method stub
                super.unload(interp);
        }


        /**
         * puts the commands to the commandtable, allowing them to be loaded by the
            interpreter
         *
         * */
        static {
                cmdtable.put("image", new ImgCmds(IMAGE, 3,3));
                cmdtable.put("resize", new ImgCmds(RESIZE, 3, 3));
        }

}
```

Listing 14.2: ImageCmds class

This class allow us to write the commands:

```
set myImage [image "\root\background.jpg" 200 200]
resize $myimage 300 300
```

Listing 14.3: Commands for Image

14.1.4 Hecl Cmds classes

The Cmds classes included in the Hecl core are:

ControlCmds - Implements control constructs like if, while, for, foreach, and so on.

HashCmds - Takes care of loading and implementing the Hecl commands that deal with hash tables, which are in turn implemented in the HashThing class.

InterpCmds - Implements various Hecl commands that deal with the state of the interpreter. For instance *set* which binds a value to a hecl variable and puts it in the hashtable, or *proc* that declares a procedure.

ListCmds - Implements the Hecl commands that operate on lists, which are implemented by the ListThing class.

MathCmds - Implements a variety of math commands such as *incr*, *sin*, *abs*, +, logical commands like or, <= etc.

PutsCmds - Implements the command in Hecl that is used to display text on the console which is the System.out in Java.

SortCmds - Implements algorithms to sort a ListThing in Hecl

StringCmds - Is a collection of methods that work with the Hecl Strings

14.1.5 Hecl Core

In addition to the Things and Commands classes the Hecl core consist of the following classes:

Command.java - Command is an interface that serves as a template for all commands implemented in Hecl. Commands takes an array of things as argument and make some calculations and calls the setResult() method in the interp class once done.

Compare.java - Compare takes either two strings or two procedures as argument. It compares the two strings or the result of the two procedures and return the result dependent on the comparison. For procedures, it will return 0 for equal, -1 if the first is less than the second, and 1 vice versa.

HeclException.java - HeclException is the exception used in hecl. If an exception is thrown when executing a script, Hecl will encapsulate the exception with a Hecl exception, alternatively throw a HeclException based on errors discovered by Hecl, and print it in to System.out together with the stack indicating where the error occurred.

HeclModule.java - Hecl module is an interface that provides for modules (Cmds classes) to be loaded by the interpreter.

Interp.java - Interp is the interpreter. It is responsible for the variables and commands that are available. Interp contains the method eval(String script) which takes the script as argument and is responsible for parsing and run the script. All Cmds-classes used by the interpreter is loaded into this class in the init of the application.

Operator.java - Operator is an interface for creating Cmds classes. It comprise the methods a collection of commands has to implement.

Parse.java - Parse is responsible for parsing the scripts into tokens. Scripts are parsed before they are executed. It is also responsible for making recursive parses and detect variable references, comment signs, calls to commands and procedures etc.

ParseList.java - ParseList is an extension to the parse class, that takes care of parsing Hecl ListThings.

ParseState.java - ParseState is the state of the current parse. It reflects whether the parse is done or has remaining characters.

Proc.java - Proc is a class that holds an instance of a procedure declared in hecl.

Properties.java - The properties class is used to parse command arguments that are set by name. A new Properties is instantiated with default properties and values, then setProps is called to substitute the default properties by the ones set in the script. At that point the rest of the command can go on, and for every property that is needed, it can be fetched with getProp.

Stanza.java - Stanza represent tokens provided to the interpreter. One Stanza is one parsed command together with it's arguments.

14.2 Running a Hecl application

As all J2ME applications Hecl requires that you have a MIDlet running in the background. This MIDlet, or canvases running on it, has to have an instance of the Interp class which is the interpreter in Hecl.

Additional commands and object may be provided to Hecl by defining them and load them into the Interp instance. When all the modules are loaded to the interpreter, the eval(Thing script) command may be called to evaluate a script.

14.2.1 Example script

In this section we will present an example script, given in Listing 14.4 and give a step by step procedure from the interpreter. Figure 14.1 shows how the parse process is executed in Hecl.

```
set Nbr1 1
set Nbr2 2
proc addTwoNbrs { nbr1 nbr2 } {
return [+ $nbr1 $nbr2]
}

set Nbr3 [addTwoNbrs $Nbr1 $Nbr2]
puts "helloworld, the sum of $Nbr1 and $Nbr2 is $Nbr3"
```

Listing 14.4: Adding Two Numbers

The eval() method first parses the entire script using the CodeThing.get() method. The CodeThing divides the entire script into characters. The CodeThing class determines whether the script is already parsed, or if it is a ListThing, which has to be parsed by a special class. If not, it creates a new instance of the Parse class, sending the script as parameter, and calls the parseToCode() method from the Parse instance. This method runs a while loop that recursively create stanza instances based on the script. The script above results in five stanzas, one for each statement. Figure 14.1 shows an action diagram of how the parse process is executed.

After the script is parsed, CodeThing returns a CodeThing instance that holds the stanzas. The Interp class then invokes the run method that runs the script. Figure 14.2 shows an action diagram of how the script is executed. The run method takes one stanza at a time, creating a Command instance from the first word of the stanza. It looks up this command through the interp class, which holds a hashtable with all the commands. Then it checks the arguments to see whether they need to be substituted, or treated in any way, before it calls the commands cmdcode. The cmdcode checks the argument length and call the Cmds class' operate method which does the computation. Our test script is parsed into the stanzas provided in Listing 14.5:

```
set Nbr1 1
set Nbr2 2
proc addTwoNbrs nbr1 nbr2 {return [+ ${nbr1} ${nbr2}]}
set Nbr3 [addTwoNbrs ${Nbr1} ${Nbr2}]
puts "helloworld, the sum of ${Nbr1} and ${Nbr2} is ${Nbr3}"
```

Listing 14.5: Stanzas From Example Script

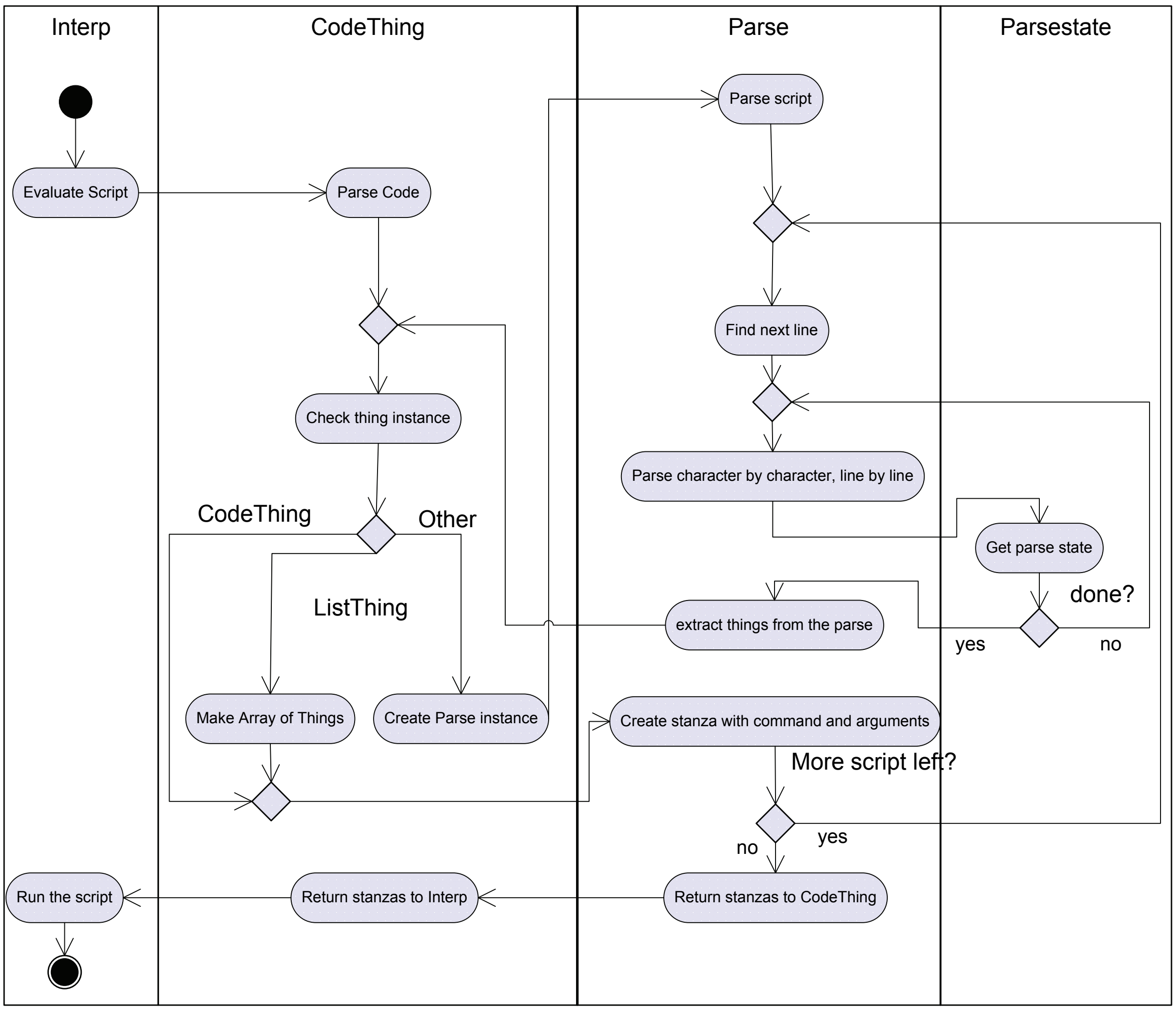


Figure 14.1: Action Diagram showing the Parsing process in hecl

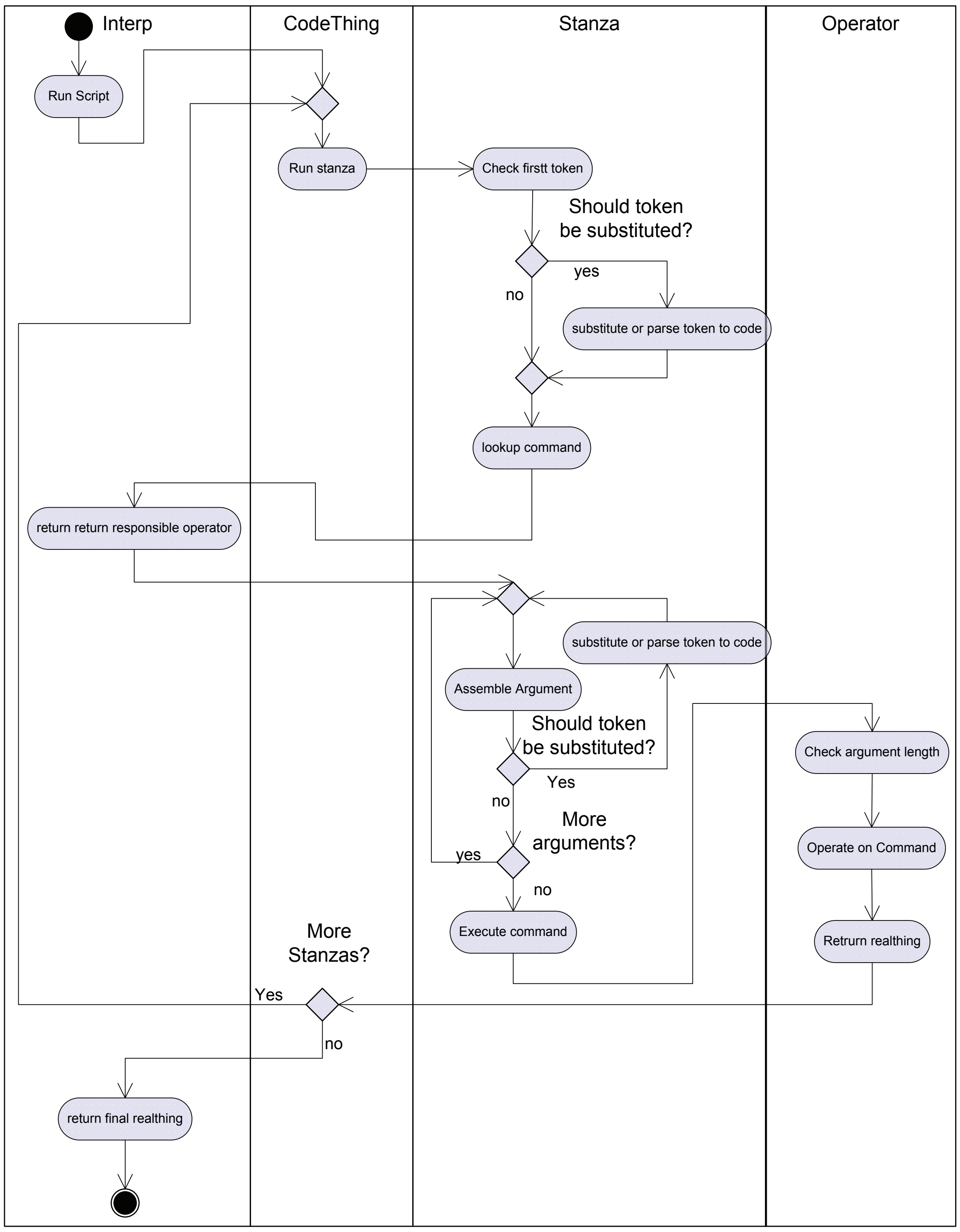


Figure 14.2: Action Diagram showing the Run process in hecl

14.3 Getting to know Hecl

Starting development of a script based client started width exploration of Hecl and it's capabilities. It took some time to get into the Hecl platform, and in the beginning we wrote a number of solutions to the language that were already implemented. The only documentation on the Hecl scripting language we have been able to find up is on it's home page, and answerers to a lot of the questions we had were not present on the site. Therefore we had to learn about Hecl's architecture and functionality and available commands mainly by hacking the Hecl core.

14.3.1 Installing Hecl

We extracted the Hecl core and installed it as a project, making us able to hack it. The first ting we did to get to know Hecl was to download the demo MIDlet posted on their site. This MIDlet contains some classes in addition to the core. These classes provides for Hecl to interface most of the basic GUI components provided by CLDC 1.1, such as forms, text field, static text etc. We went on to make our own scripts, and made some simple programs that took a string as input, manipulated it and displayed it.

14.3.2 Hacking Hecl

As we began to understand the syntax we wanted to take a look under the hood of Hecl to figure out how to manipulate it. First off we made Hecl compatible with some functionality it lacked, which we thought was important to support the animation. This functionality was firstly support for random generators. We achieved this by hacking the MathCmds file. The J2SE version of Hecl support's Random number generation, but for some reason it has been excluded in the J2ME version although J2ME supports it. Listing 14.6 shows the code added to the MathCmds file in the hecl core to support random numbers generation.

```
public static final int RAND = 33;
...
case RAND:
        return NumberThing.asNumber(new Thing(Integer.toString((new Random()).nextInt())));
...
cmdtable.put("rand", new MathCmds(RAND, -1, -1));
...
}
```

Listing 14.6: Adding Random Number Generation In Hecl

The random generation of numbers seemed to work fine, and we began to realize how easy it was to extend the functionality of Hecl. We moved on to extend the functionality of Hecl to support threads. We thought that this functionality would be convenient to use in animation context. We made the command *thread* in the ControlCmds file, taking code to be evaluated as parameter. In addition to this we made a single Hecl command *sleep* to allow for code to pause for a given interval. The thread also seemed to work fine on the testing we did in the beginning, but we encountered some problems due to the processing overhead later on when testing more advanced programs, we will come back to these problems.

14.3.3 Adding GUI Support

At this point we wanted to provide control of the Canvas' Graphics component in Hecl. The CLDC 1.1 version of Hecl support the standard GUI components provided by java, but it does not support drawing. The game controllers we had developed to our first implementation were made by drawing primitives and images on the canvas class, as well as using GUI components provided by the ServiceFrame framework. The GUI support in Hecl 1.1 (the CLDC version) is an extension of the core. All we had to do to extend the class GUI, which holds the GUI commands part of the interpreter, to support painting was to add support for drawing was to make constants representing each primitive and give the GUI class control of the Graphics object of the canvas and draw them.

GUI support for the scripted client were first out supported to the GameCanvas class that comes with the J2ME library. This solution provided the GUI class with the main gameCanvas's Graphics object. For every statement in the script containing a paint command, the Graphics object were called with the values it needed. However, we have had some problems width the GameCanvas class due to it being double buffered. This means that it as a contrast to the Canvas class, buffers two screens in the memory causing it to use's twice as much heap as the canvas class. We changed the main MIDlet to extend Canvas instead of GameCanvas. This was convenient since we did not use any of the extra features provided by the GameCanvas class in the first place.

The Canvas class do not directly give control of the Graphics object to the other classes, but require them to call the repaint method. The repaint triggers the paint method which determine which elements to paint. Our first solution was to call the interpreter to calculate the paintscript for every time the repaint method was called. We had some problems with this because of the overhead generated from the interpreter. This overhead was critical with animations that required a repaint every hundred millisecond. Therefore we made specific paint objects that were calculated once from a specific paintscript, and repainted on demand. The primitives that should be drawn was stored in a Vector as paintcomponents. This solution minimized the overhead a bit, but not enough. Besides, this method had a problem with attributes tied to the paintcomponents. Binding paint objects x- and y- location to a value in the script worked ok, since the interpreter tied that value to the hashtable containing all program variables. However, binding this value to a dynamic expression did not. The expression return one immediate value, which were stored with the paint object on initiation, rather that a reference to the expression which were calculated on every call. To find the weakest spot generating most overhead we used the Sony Ericsson emulator's profiler.

The profiler showed that 28 percent of the CPU time was used on the instantiating of images, and that approximately 40 percent were used on the Images resize method.

Knowing this we saw the need to instantiate the pictures to be used in the game before they were used (painted). To be able to do this we had to add support for pictures in Hecl. We did this by adding new classes to the core as showed in Listing 14.1 and Listing 14.2.

We added a new class called ImageThing, and a related class with commands to operate on it called ImageCmds. This way we was able to instantiate the images before they were used, and the ImageThing instances were sent as parameters to the paint object that held a draw Image command. This introduced another problem, because, although the variable were changed, the reference within the paint object did not.

We figured that at this point we would have enough functionality in Hecl to make a test version of GUI from the game client we developed during the previous project. All logic, such as animation, was written in Hecl. We used our thread implementation to make the animations. This did not work out quite as well as we had pictured it. One animation that were supposed to display forty pictures managed to show about three to four. This was a result of the paint script being calculated every time the repaint() method was called. As the sleep interval got smaller, to about 30 to 50 milliseconds, the repaint just did not manage to calculate the paintcomponents through the script, and the call to repaint() was neglected. We tried using the serviceRepaints() method, that insures that repaint() is being executed, but without any luck as it resulted in a slow animation. When the sleep interval became even smaller, the script did not executed at all because the script did not get time to read.

We was stubborn enough to rewrite solution for this about ten times, wasting much time. We alternated whether the script should be initiated once or whenever the call to repaint was called. Whenever we made the script being initialised from the start, storing all the primitives to paint in a Vector or Hashtable, we found a source for overhead. Eliminating this point made us go back to the solution where objects were painted on demand. No matter how much overhead we managed to get rid of, the paint on demand solution did not seem to work well with animations. After a couple of weeks stumbling in that circle we finally decided that the paint scripts had to be initialized from the start. This meant that we had to make the attributes to the components dynamic.

Forunatly, the rest of the scripting platform for MOOSES game controllers development went more smoothly. At this point we had more experience with hecl and limitations it caused.

Part III

The MOOSES Scripted Client Architecture

CHAPTER 15

Architecture Considerations

15.1 Architecture Background

Software architecture is a field of study that is becoming more important every day. As systems become more large and complex, the difficulty of meeting requirements increases dramatically. The software architecture discipline is centred on the idea of reducing complexity through abstraction and separation of concerns. By achieving this at an early stage, the system is more likely to meet its requirements since it allows for reviewing design issues before implementation, reducing risk and costs. In this aspect, software architecture is of high importance for this project in hopes of avoiding a poor, low-quality, non-flexible and over-complex framework. This Part contains information regarding system stakeholders, quality attributes associated with the architecture, architectural tactics[1]- and patterns[2]-choices made to achieve quality requirements. The practices used here are adapted from the *Software Architecture in Practice*, by Bass, Clements and Kazman [54].

15.2 Stakeholders

In this section we will try to explain who the different stakeholders are for the system, which interests they have to it and what views that are important for them. The views are covered in .

[1]Architectural tactics: Known methods for achieving quality in a system

[2]Architectural patterns: A description of element and relation types together with a set of constraints on how they may be used

Developers: This project is part of a bigger concept that includes three developers. The concept also requires other third party developers to write software for it. And allow for extensions or alternative ways for usage which may involve other developers. All the views presented in this document are important for developers to provide a better understanding of the ideas behind the framework.

Maintenance: If the MOOSES framework is distributed, it will require maintenance which may be done by a person sitting with the appropriate knowledge. The maintenance is not likely to include works on the client, but a person that perform maintenance or repair on an implementation of the framework will have an advantage of knowing the functionality of the client, since it may be the cause of an error. The physical view is important, together with the documentation on the code to better understand the layout and mode of operation of the client in relation to the concept.

Financial: This project is based on cooperation between NTNU and Tellu. Both of these actors, together with the developers, have financial interests in the project.

End users: The users using a MOOSES implementation may be divided into players and administrators. Administrators may find parts of the process view interesting, together with the physical view.

CHAPTER 16

Requirements

This chapter will present the functional, non-functional and environmental requirements of the scripted client solution.

16.1 Functional requirements

This section represents the functional requirements of the scripted client implementation. Functional requirements are a set of instructions reflecting the functionality which must be implemented in the application. The requirements are presented in Table 16.1.

Additional information about the functional requirements

FR 3 The developer should be able to load wav files and play them by calling a statement from the script.

FR 4 The developer must control code to be evaluated if a key is pressed or released. This includes an interface for telling whether or not the key should send a keyPressed or keyReleased message to the server. The developer must also control whether or not a key should repeat while being pressed, and how frequent it should be repeated.

Functional Requirements	*Description*
FR 1	The implementation must support graphical components as primitives and pictures
FR 2	The implementation must support dynamic changes in graphical components attributes
FR 3	The implementation must support sound feedback
FR 4	The implementation must support logic for user input
FR 5	The implementation must have support for the most common animations
FR 6	The implementation must support multiple canvases
FR 7	The implementation must support interface for communication from and to the server
FR 8	The implementation must support interface for independent logic such as threads

Table 16.1: The functional requirements for the scripted client implementation

16.2 Quality Requirements

The non-functional aspects are an important thing to keep in mind when designing applications. Non-functional requirements include constraints and qualities [55]. Qualities are properties or characteristics of the system that its stakeholders care about and hence will affect their degree of satisfaction with the system. Constraints are not subject to negotiation and, unlike qualities, are (theoretically at any rate) off-limits during design trade-offs.

The non-functional requirements are not so clearly stated by the users and stakeholders of a system, but are nonetheless important for the user satisfaction of the architecture. The requirements will be presented as ways to achieve the quality attributes: Usability, Performance, Modifiability, Availability, Security and Testability. The following roles are mentioned in this section and it is important for the reader to distinguish between these:

- **The framework developer** - The developer that is responsible for changes and modifications to the framework and the script part of the client framework core. This group is interested in the quality requirements: Modifiability, Testability
- **The game developer** - The developers are responsible for writing game controllers that run on top of the framework. This group is interested in the quality requirements: Testability
- **The user** - The persons playing the games running on the framework designed by the game developers. This group is interested in the quality requirements: Usability and Performance
- **The system owners** - The organization / person(s) running and maintaining the

framework and games. This group is interested in the quality requirements: Modifiability, Availability and Security.

16.2.1 Availability

A systems faults and failures are associated with availability. A fault occurs when something does not go as intended and is not visible. For instance, if a data package is sent from one of the devices without one of the devices registering the loss, a fault has occurred. If the device that sent the package acts like the other device has received the package, a failure might occur.

A1 - Packet loss	
Source of stimulus	Runtime issue
Stimulus	The game server sends a status message to the client
Environment	Run time
Artefact	The game server module
Response	The client synchronize with server
Response Measure	The game client synchronize with server once a minute

Table 16.2: A1 - Packet loss

16.2.2 Modifiability

Modifiability mirrors changes to the system. It is vital that changes may be performed without too much hassle. For instance, if a famework developer wants to add new commands to the interpreter, this should not involve changes to more than the module that handles the interpreter. A change does not necessarily need to be made by a maintainer or developer. It can also be made by the end-user, for instance a configuration set-up.

The client is designed in a way that allows future modifications and/or additional modules.

M1 - Extend the interpreter with additional commands	
Source of stimulus	The Framework developer
Stimulus	The framework developer wants to extend the commands supported by the interpreter
Environment	Design time
Artefact	The mobile client
Response	The framework developer provides the additional Cmds- and corresponding classes to the client JAR file, and makes the interpreter load the new modules.
Response Measure	The developer does not make any modifications to other parts of the client

Table 16.3: M1 - Extend the interpreter with additional commands

M2 - Change dynamic game related data	
Source of stimulus	The user
Stimulus	The user's actions make changes in the local variables
Environment	Run time
Artefact	The mobile client module
Response	The controller canvas runs the script associated with the action
Response Measure	Dependent on how intuitive the interface is, and the complexity of the changes, such a change should be doable in a couple of seconds.

Table 16.4: M2 - Change dynamic game related data

M3 - Extend the framework with more states	
Source of stimulus	The Framework developer
Stimulus	The framework developer wants to extend the domain of the framework to fit other concepts.
Environment	Design time
Artefact	The mobile client state machine module
Response	The framework developer has to add new states to the state machine running the client, and determine how the client should act on messages received in the different states. He may also remove redundant states. If the state requires additional corresponding classes, these must be added.
Response Measure	Manipulation of the backbone state machine should be possible without side effects.

Table 16.5: M3 - Extend the framework with more states

M4 - Adding new games to the client	
Source of stimulus	The Maintainer
Stimulus	The maintainer wants to add more games on an implementation of the concept.
Environment	Maintenance time
Artefact	Game server module
Response	The Maintainer adds the game to the server together with the corresponding script and client resources.
Response Measure	Adding new games is done without interfering with any code or modules

Table 16.6: M4 - Adding new games to the client

16.2.3 Performance

Performance mirrors the systems ability to respond to an event that occurs and time to execute it. Such events may come from several instances. These instances can be an end-user, the system itself or from other systems.

P1 - Response time from player actions	
Source of stimulus	The User
Stimulus	The user gives input to the game through his/her controller (mobile)
Environment	Run time
Artefact	Script interpreter
Response	The action chosen by the user should play out on the screen(s).
Response Measure	If the input involves scripts to be executed, it should take no less than 0.5 seconds, independent of underlying hardware.

Table 16.7: P1 - Response time from player actions

P2 - Fair premises	
Source of stimulus	The client
Stimulus	The client runs on a resource constrained device
Environment	Run time
Artefact	Script interpreter
Response	The gameplay experience should not be affected by the limitations of the client
Response Measure	The client should be efficient enough to run on all the clients supported by the former client

Table 16.8: P2 - Fair premises

16.2.4 Security

Security is concerned with the systems ability to prevent unauthorized usage/access without compromising normal usage. Attacks can be unauthorized attempts to access or modify data.

S1 - Player tries to cheat	
Source of stimulus	The User
Stimulus	The user tries to alter the script that represents the game controller
Environment	Run time
Artefact	Game mobile client module
Response	The client downloads a new script whenever a game is started, or a user logs on.
Response Measure	Depending on the size of the script. Average size is a couple of Kilo bytes that should be distributed in one or two seconds.

Table 16.9: S1 - Player tries to cheat

16.2.5 Testability

To find bugs, faults and leaks in the system, it needs to be testable. Designing an architecture that can easily be tested for faults will save a lot of time. There are several different ways of doing this. Most of them involve monitoring the systems internal state and outgoing output that is easy to interpret.

T1 - status of running modules	
Source of stimulus	The System owners
Stimulus	The system maintainer wants to check the status on the running system.
Environment	Maintenance time
Artefact	The Framework
Response	The framework must supply a GUI based tool for debugging of the system. This GUI must also profile attributes related to the scripted client, and the game running on the game server
Response Measure	The debug GUI should respond with the current running status of all modules currently running in the system.

Table 16.10: T1 - status of running modules

T2 - Test a new game-client script	
Source of stimulus	The Game developers
Stimulus	The game developer wants to test the controller they have scripted for their game.
Environment	During game development
Artefact	Editor tool
Response	The Framework must include tools for writing and testing scripted controllers. Such a tool should also be able to profile the program and display the measurements in wanted format.
Response Measure	The Script editor should provide profilers and feedback on measurements.

Table 16.11: T2 - Test a new game-client script

T3 - Game developers wants to test their client	
Source of stimulus	The game developers
Stimulus	The game developers test their client without having to run the game
Environment	During game development
Artefact	Editor tool
Response	The editor tool must simulate server connection, and provide an interface for simulations.
Response Measure	The game developer is able to test the scripted client without running the game.

Table 16.12: T3 - Game developers wants to test their client

16.2.6 Usability

Usability is concerned with the learning curve, how easy certain tasks may be performed by the user and how intuitive the system is for the user in the way it displays information. Usability is an issue that often must be considered in the early stages of architectural design. If a major problem related to usability is detected late in the project phase, the repair and modification that have to be done to the architecture will cause more work than if these problems are illuminated in the architecture design phase.

U1 - The user wants to exit the application	
Source of stimulus	The user
Stimulus	The user wants to exit the application
Environment	Run time
Artefact	Mobile client
Response	The procedure for quitting the application is the same in every part of the application, independent of the scripts
Response Measure	The Exit procedure involves two key-presses from the user, and an exit protocol that takes a half to one second depending on traffic.

Table 16.13: U1 - The user wants to exit the application

U2 - The user wants to log in to the system	
Source of stimulus	The user
Stimulus	The user wants to log in to participate in the games
Environment	Run time
Artefact	Mobile client
Response	The user should only concern about starting the application and type in desired nick name. Any other actions should be taken care of by the state machine.
Response Measure	The user may log in without any advanced setup

Table 16.14: U2 - The user wants to log in to the system

16.3 Environmental Requirements

In this section we will give a short description of the environment that is needed to run the framework.

Java Our framework is based on the newest Java technology. Therefore the devices must support the CLDC 1.1 configuration and the MIDP 2.0 profile.

Java Bluetooth API The devices must have the Java API for Bluetooth to make the Bluetooth device accessible from Java.

Bluetooth The devices must support Bluetooth 2.0 to provide the bandwidth needed, and preferably class 2 Bluetooth to expand the covering area and to reduce probability for packet loss.

Sound It is desirable, but not crucial, that the phones support the Advanced Multimedia Supplements (JSR 234) standard to be able to give sound feedback on the phone.

Hardware Phones using a faster processor may get an advantage not only by obtaining faster communication with the server, but also by providing a better GUI that can be refreshed faster. For this reason we have put a lower threshold on the phones supported. So far we have only worked with mobiles manufactured by Sony Ericsson, and the threshold will therefore be set at the phone with the oldest phone which passed the test. Another factor that speaks in favour of the other devices is that the Sony Ericssons mobiles are leading on in the size of the memory heap. Our application consumes a god slice of this heap (between 300 to 1000 bytes depending on resources used and operations on them). We have set the lower threshold to SE K750 which gives a little disadvantage compared to the newer phones when running on our framework.

CHAPTER 17

Design Decisions

This chapter discuss many of the design decisions made on how different parts of the new client implementations should be implemented, together with a brief presentation of the some of the decisions we made one the previous implementation that ate still relevant.

17.1 Tellu ServiceFrame

ServiceFrame was a perfect hit for our base system because their platform support for mobile clients, servers and multiple ways for communication between these. It allows the state machines to be placed anywhere in the deployed system (At different servers) without changing addresses. This means that we may move the state machines to higher level servers and down again without any big impact.

17.2 State machines

When we designed the architecture for the MOOSES framework, we wanted it to be as flexible and quick to design and implement as possible. We had a look at the Finite State Machine (FSM) technology and found it to suit our needs to provide good extensibility, and being fast and maintainable.

The MOOSES client comprises five states. The MOOSES state cycle is presented in Figure 17.1. The two first are initial states, while the client will cycle between the three other

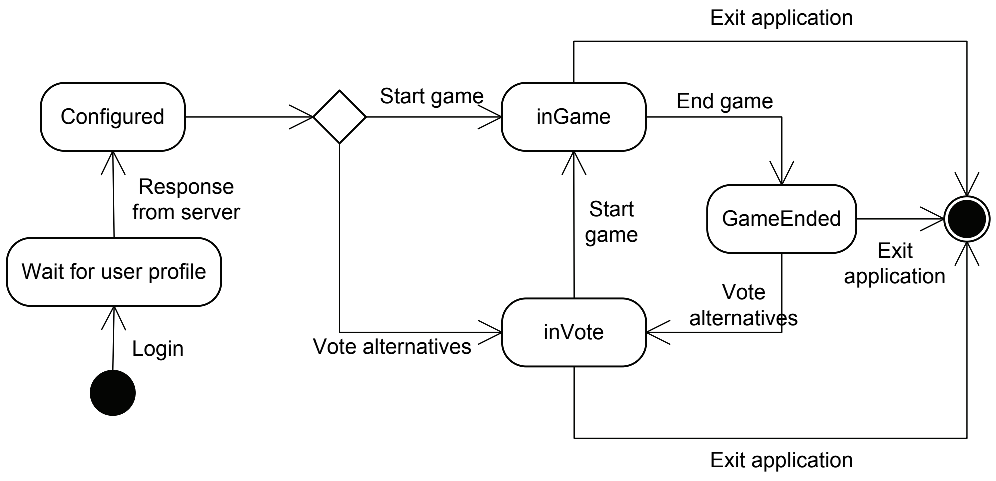


Figure 17.1: MOOSES client states

states on runtime. The first state registers and authenticates the client with the server. If the authentication is validated, the client will go to configured state to wait for the server. The client will receive either a startgame message or a vote alternatives message, and go to the corresponding state. From this point, the client will cycle in this order, inVote-inGame-endGame-inVote.

17.3 Message System

The communication between clients and servers in ServiceFrame follows a message system. Each message extends a default interface and adds the additional functionality needed by the specific message. The message is serialized during transport, and deserialized when arriving to the receiver. The state machines use these messages to change between states, and to distinguish information between them self. The state machines use the *instanceof* check in Java to differentiate between the incoming and outgoing messages, and perform the necessary actions.

Using these messages comes with advantages and disadvantages. The messages make it easy to route information within the state machines, and to bind functionality to the data. However, since each message has to be declared in its own class, it extends the size of the client for each message added. Also, adding new message classes requires manipulation of the statemachines source code. This would not be appropriate when using a scripted client that does not allow changes in the core. Therefore, the only messages used in the new implementation (the scripted controllers, not the entire client) is the KeyMsg that sends a keypress or keyreleased message to the server, and a VibrateMsg that allows the server to trigger the Vibrate function on the phone. All other communication between the Game and the controller is done using an AFPropertyMsg provided in Appendix D. This contains a Hashtable and the ID of the message as a string. This solution is convenient because it makes it easy to transport data from and to the script.

For instance, in the test game for the previous implementation we created a ShootMsg to tell the server that the client had initiated a shoot. The message contained an integer value indicating the weapon the client used. With the new implementation the message is sent from the script. The ID of the AFPropertyMsg is "ShootMsg", and the integer is stored in the hashtable. StatusMsg from the previous implementation that provided the client with the current status in the game with respect to health and score has also been replaced by an AFPropertyMsg containing the same values. These values are easy to retrieve in the script and may be used as the author of the controllers desire.

17.4 Script code

The scripted applications use a predefined template. A scripted game controller may comprise several canvases. Each canvas is declared using the canvas command, followed by the name of the canvas and a tag *begin{* and closed by a tag *stop}*. Multiple parts of the canvas is distinguished by the ! token.

Listing 17.1 shows an example of how a canvas is created for our scripted client. More examples are provided in the Appendix A.

```
canvas maincontrol begin{

! main
# initate some values
set frags 0
set maxammo [list 13 30 4 8 1 ]
set bg [image "/res/background.jpg" $reswidth $resheight]

! paint
#draw the background picture
paintpicture src $bg x 0 y 0 "left|top"
#draw a rectangle
paintrect x 0 y 0 width {[- $frags 1]} height [/ $resheight 3] depth 3

#input code for the 5-key
! 5
presscript: set frags 0

! UpdateMsg
#input from the server, extracting the values and replace the local
set frags [hget $msgHash frags]

! shownotify
# executes when the canvas is loaded to the screen
set frags [lget $maxammo 3]
stop}
```

Listing 17.1: Scripted controller example

The ! main token represent a section where the variables used by the script is declared, it is executed instantly by the underlying Java canvas and stored in hashtables. The ! paint token is the canvas paintscript. It comprises several paintcomponents that are stored in a hashtable and redrawn each time the flush command is called from the script or an animation triggers a repaint event. By giving parameters to a paintcomponent enclosed by brackets, for instance *height {$height}* , the height parameter will be dynamic and point to a variable in the script or calculations that are evaluated on draw. Both these tokens are immediately executed in the init, their values stored in hashtables, and then the scripts are discarded. The rest of the tokens are pre-parsed, and stored as CodeThings in a hashtable in the underlying canvas.

CHAPTER 18

Design Overview

Now that we have established the requirements for the client, it is convenient to get the base design down to help getting a rigid and easy-to-maintain system. The design overview may help to wipe out problems at an early stage, and provide for other actors to get a better understanding of the systems inner and outer workings.

This chapter will focus on the design views of the scripted MOOSES controller architecture, and some views of the complete framework to give a better understanding of the entire concept.

18.1 High Level Architecture

The client architecture is module based with separate packages for each major type of classes. The top level structure of the client architecture is provided in figure 18.1. This is to ensure that certain parts of the client may be redesigned, replaced or tailored for specific needs without rewriting the entire client.

The client application comprises one MIDlet that runs different parts of the application based on the state of the state machine.

State Machine The state machine is the backbone of the client application. It serves as the clients face outwards in relation to the game-server, and decide which module that is loaded to the display at different times based on messages from the game server.

Default Window The default window is loaded when the client is idle waiting for the server. This occurs on initializing states and when games end.

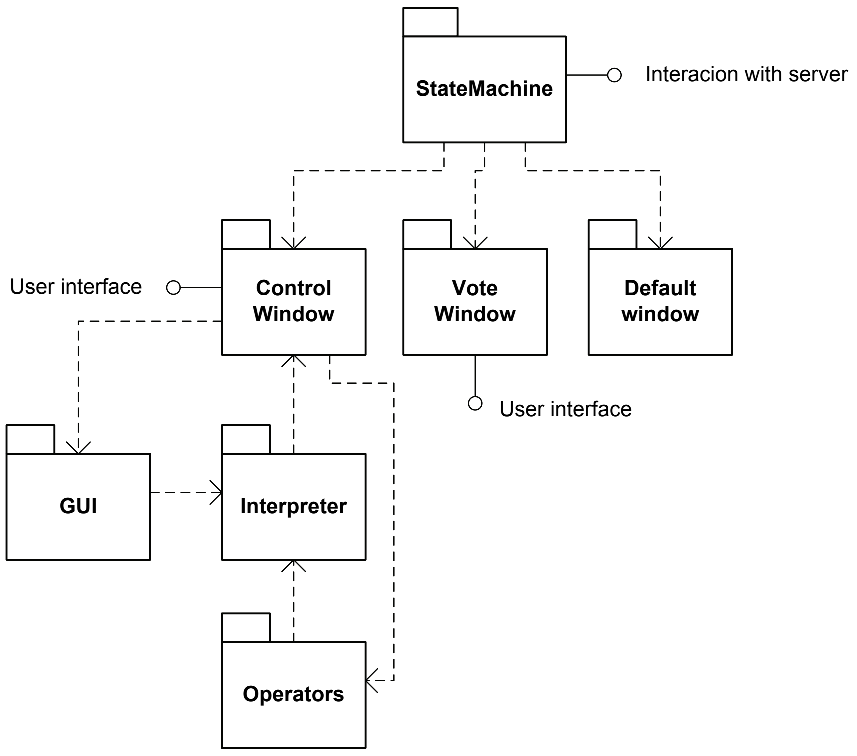


Figure 18.1: High level architecture for the scripted MOOSES client

Vote Window The vote window is the program loaded to allow the user to participate in a vote session. The server provides the client with available games, that are put in a list on the vote window. When user selects a gane, a vote is sent to the server.

Controller window The controller window is the application that let the user interact with a game. This window has interfaces that connect the script and the game server, through the state machine. It runs the interpreter that executes the scripts. When a game is started, a new controller window is created with the script for the game controller as argument. The game controller divides the script and stores the information about the game controller application, provided by the script, in hashtables. The game controllers are built up by different canvases that run on top of the controller window. The controller window contains most of the data constitute the canvases, to reduce the scripting computation needed.

Interpreter The interpreter is the interface between the script and the Controller window. The canvases that run on top of the controller window comprise script fragments that are evaluated through the interpreter. The interpreter also contains the variables used by the scripted game controller application.

Operators The operators are parts of the interpreter that handles different commands. This include commands for logic, input, sending and receiving data, sound feedback, animations and operations on data types and objects. When framework developers add a new commands to the framework, they extend the operator and load the new operator into the interpreter.

GUI The GUI module is a special operator that computes a paint script, and produce collections of paintcomponents that are associated with the different canvases. These collections are stored in the GUI module, and fetched on demand by the controller window when changing between canvases.

18.2 Data flow view

To get a better understanding of the communication between the actors involved with the client we will provide a Flowchart diagram. Figure 18.2 shows the dataflow in the Scripted MOOSES client.

External Actors The external actors are elements that provide data to the system, and that receive the data produced by the system. The user provides the system with user input in form of key presses and key releases. The system provides the user with information by refreshing the display, to reflect the current situations.

Triggers are elements in the scripted solution that executes in their own thread. The Hecl is designed to run code once in the underlying Java MIDlet thread. Triggers run in their own thread, and may periodically, or on demand, perform logic. Code may also be executed from the script by receiving data or by user input.

The game server hosts the games, and the voting sessions. The server tells the client which state is supposed to go to, and provides the client with data from the game.

Start game The client will go to inGame state when receiving a startGameMsg, containing the name of the game. If the client does not have the script it will request the script and the corresponding resources from the server. The script will be divided into fragments, and stored as collection of data for the different canvases on the controller window. The game will be loaded on the client, and the user is ready to play.

Refresh screen The display is the user's local reference to the game. It reflects the data that is relevant for the user. Changes in variables will result in changes in the representation, either animations, changing canvas or other representations.

Send Data The developer controls protocols for sending data to the server. Sending data is a result of user input, an event from a trigger or as a response for received data. The data collection is a hashtable that contains Java Objects. The hash's and name of the message has to be the same on the game side and the Java side.

Receive Data The scripts contain tags for receiving data. These tags include code to execute on the data received.

Perform computation Different actions may have scripts attached to them. This computation may result in local changes, communication with the game or both. Computation may also create triggers or trigger other script parts.

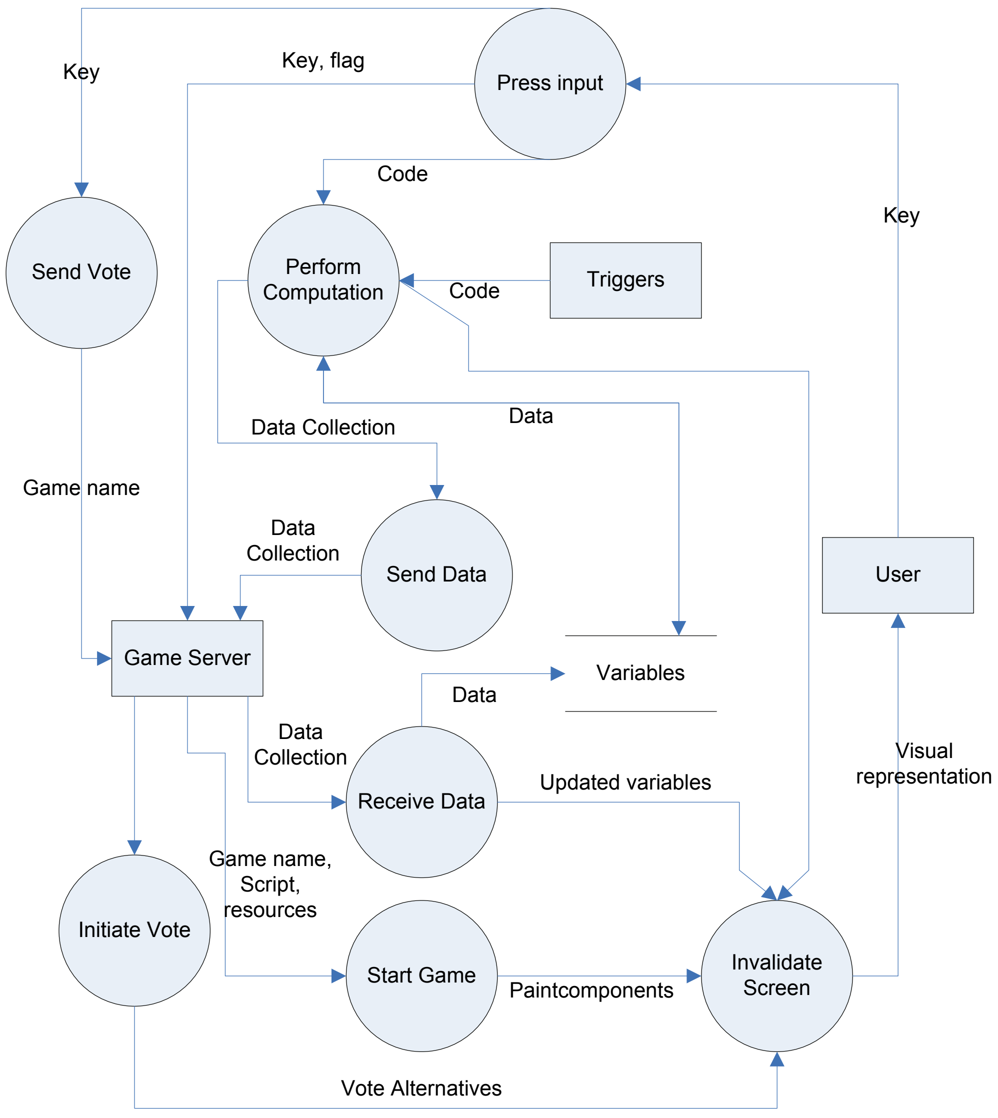


Figure 18.2: Dataflow diagram for the scripted MOOSES client

Press input The key input is the users interface for interaction with the game. Pressing or releasing a key may result in a key message to the server indicating the key, and a flag indicating whether the key was pressed or released. User input may also trigger scripts to execute.

Initiate Vote When the game server sends a voteAlternatives message, the client will go to vote state. The message contains all the vote alternatives, which is put in a list. The user chooses a game, and sends his vote. The vote module of the MOOSES client does not concern the script module.

Send Vote When a user sends his vote, a vote message is sent to the game server, containing the name of the game the user voted.

18.3 Process View

The process views give a better understanding of how the different components run and relate to each other in real time. This will dredge the way communication occurs between the modules and the functionality of the modules.

18.3.1 Joining games

When a user is connected to the game server, he will join the game that is running on the server. The server sends a startgame message containing the name of the game. The client responds with a resource request message if it does not possess the script. The process of joining a game is provided in figure 18.3

After receiving the resources, the state machine instantiates a new Controller window with the script as argument. The controller window first loads all the modules to the interpreter, then proceed with dividing the script. The script is first divided on different canvases. A canvas is identified with a name, and a body that contains all the scripts related to the canvas. The scripts are further divided into tags. The main and paint tags are executed directly and then discarded. Running the paint-tag will result in a list of paintcomponents that constitute the visual representation of the canvas.

The other scripts that constitutes a canvas is input , data scripts and scripts to execute when a canvas is being displayed or hidden. The controller window stores information for each key as a collection of press and release scripts, keyrepeat options and flags that indicates whether or not the key should send a keymsg to the server. This information is stored in an instance of the inner class Input. The Input instances are further stored in the hashtable which possess all the information about a canvas that concerns the controller.

The data scripts are used for receiving data. The tags of these scripts are called the same as the message they represent, and they will be parsed and stored in the canvas' hashtable.

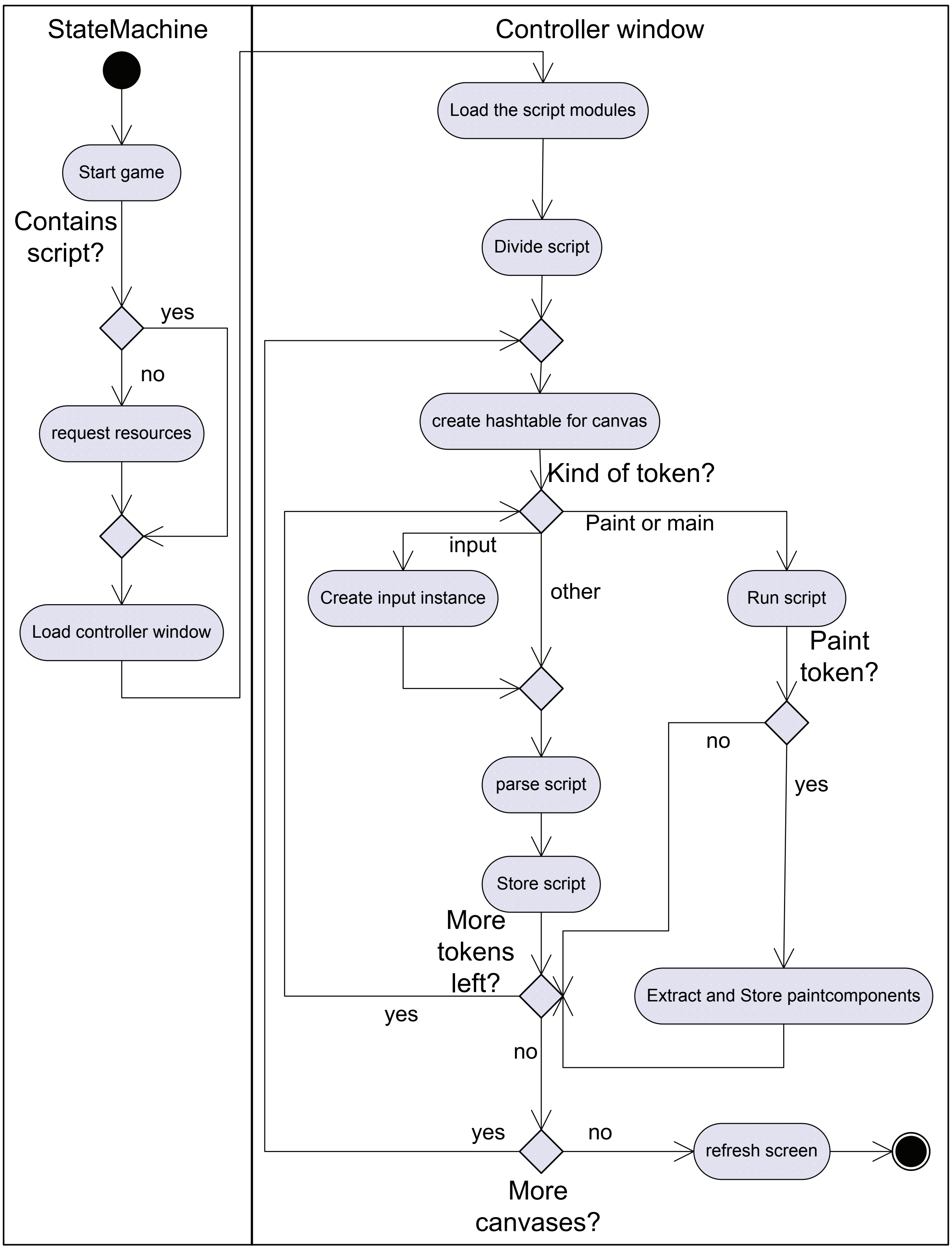


Figure 18.3: Process view for initiating games on the scripted client

When all the canvases has been stored in the hashtables representing them in the controller window, the controller window is set to point at the first canvas from the script, and the repaint() method is called. The screen refreshes, and the player is free to play.

Part IV

The Scripted MOOSES controller development framework

CHAPTER 19

Development of a script solution

During this master thesis we developed a prototype implementation of the scripted client based on the Architecture given in Part III. This chapter will present how the prototype was written with respect to changes done to Hecl and additional code and functionality added. It will also provide information on the features available for developing game controllers for MOOSES with the scripting platform.

19.1 Manipulation of Hecl core

To tailor Hecl to our needs we had to make some changes in the existing framework. Some classes where added to the Hecl platform, but manipulation of the core was necessary as well. The changes, mentioned previously, to give thread support and random number generation in Hecl where made directly in the Hecl core classes.

19.1.1 If statements

Hecl does not provide a thing for the boolean variables. Boolean variables are represented as an IntThing with zero as false and one as true. This created a problem, because "true" or "false" where not capable as arguments, only the calculation of boolean expressions. The reason for this is that the IntThing's isTrue() method throws an exception for any thing that is not a NumberThing, meaning that "true" that might be a StringThing may raise an exception. We changed the if command in the ControlCmds class to accept both positive numbers and "true" as true.

19.1.2 Expressions

Since all the drawing operations are done on application start-up, and never called again, we needed for the paintcomponents to accept dynamic arguments. We based our GUI implementation on the one provided with the CLDC version of Hecl. This class calculates all the arguments in advance, and they become permanent. We changed the GUI class to provide an array of paintcomponents to every canvas. These paintcomponents comprise primitives, images and animations.

Example

```
set height 200
set drawrect true
paintrect x 100 y [/ $height 2] widht 100 height 100 fill false dept 4 draw $drawrect color
    red
```

Listing 19.1: Example Using Expression as argument

The line *paintrect* above creates a new paintcomponent that draws a red rectangle with 4 pixels thickness at point (100, 100), 100 pixels wide and 100 pixels height. Arguments such as fill, draw and dept may be neglected, and the default values will be loaded. All the arguments are calculated once in advance, and remain as single values for the rest of the runtime. Changing $height or $drawrect will not affect the paintcomponent.

We chose to initialize the paintcomponents only once, with the opportunity to manipulate the arguments later on. The final version of the dynamic argument implementation sends the argument collection as parameter to the paintcomponent. When the paintcomponents draw method is invoked, the arguments are calculated for every call. This is where we first encountered the expression problem. We used the Interp's eval(Thing t) method, which is supposed to parse and evaluate a statement, or a collection of statements. This method is also used recursively to evaluate expressions in Hecl. Despite the fact that the method returns a thing indicating the result, it would not allow us to use the method to evaluate the dynamic arguments. We had to manipulate the Interp class to provide a method for evaluating expressions. Most of the manipulation was done in the Stanza class. A flag in the Stanza class indicates whether it is an expression or statement that is to be calculated, if the flag is set, Stanza will return on a certain point with the calculated value.

19.1.3 Pre parsing

When the controller window divides the script based on canvases, it parses the scripts constituting the canvases into CodeThings. A hashtable contains the fragments of canvases that constitute the application. Different parts of the script are called in different contexts. For instance, the current canvas might have its own scripts for input. The pressedkey script will then occur when a user presses a key. Interp's eval(Thing t) method will be called with the script to evaluate. This produces a significant overhead that can be reduced. The overhead is a result of the Interp having to parse the script for every time. Hecl is designed to load a script once, not occasionally like our implementation does. If the scripts are parsed only once under the initiation process, we could more than halve the overhead. The parse operation will result in a CodeThing containing the parsed script as a collection of Stanzas. The Interp will not parse a CodeThing, just evaluate it. We made an interface in the Interp class for parsing the scripts without executing them.

19.1.4 Global statement

By default, Hecl always use local variables in procedures. To use a global variable, the global statement has to be called to load the variable into the scope of the procedure.

Example

```
set Nbr1 1
set Nbr2 2
proc getSumOfNbr1AndNbr2 {} {
global Nbr1 Nbr2
return [+ Nbr1 Nbr2]
}
```

Listing 19.2: Hecl global Example

Calling global in the example above will allow the global variables to be used inside a procedure body. However, the global variables are read only, meaning that any changes done on the variables in the scope will not affect the values they represent. We figured we did not need for global variables to be loaded into the scope, and certainly not read only variables. Therefore, we removed this feature from the Hecl core. The prototype makes all variable declarations global not only for the current canvas, but for the whole scripted controller application. This means that variables that belong to GUI components or other data may be edited from independent from applications state.

19.2 Variables

Declaration and usage of variables that contains primitive types in the scripted client is done by the standard in Hecl. Declaration of variables that contains objects on the other hand is done by using the operators (Cmds classes) for the specific object. The operator also handles statements that operate on the object.

19.2.1 Constants

Some of the variable names are locked to provide some constants useful for computations. From the core, Hecl provides constants for mathematical PI and mathematical e. Our extension of the Hecl framework provides the constants *$reswidth* and *$resheight* that reflects the width and height of the mobile devices screen, providing for developers to make dynamic GUI.

The variable name *$msghash* is also dedicated to the development, although it is not a constant. The underlying controller canvas use this variable to dispatch data received from the server to the interpreter, making it available through the script.

19.3 Additional functionality

In addition to the changes done to the core of Hecl, we added commands for the elements needed to develop a game client for MOOSES that supports scripted controllers. Development of these features where done by providing classes in Java and Cmds classes to work on them.

19.3.1 Graphical Elements

The CLDC 1.1 version of Hecl comes with a module for basic MIDP GUI components, but it has no support for drawing primitives or other graphical elements. The MOOSES control client is based on the MIDP Canvas that draws directly on the canvas through a defined paint() method. This implementation also supports the GUI elements from the ServiceFrame framework.

We figured we could use the GUI class distributed with Hecl, but we had to rewrite it to suit our purpose. As mentioned earlier this drained most of the development time, pending between different solutions to optimize the implementation. The final implementation of GUI support for the scripted client requires the paint scripts to be loaded at application start-up. Running the script will result in an Array of PaintComponents that are stored and associated with the canvas.

The paint script is separated by the "*! paint* " tag, and consist of a collection of paint statements. For instance:

```
paintrect x 0 y 0 width $reswidth height $resheight fill true color red
```

Listing 19.3: Paint Component Command

The code-line in Listing 19.3 tells the paintscript to fill a red rectangle in the entire display. The variables $resheight and $reswidth mirror the width and height of the canvas. These are predefined constants that are necessary for dynamic accommodation of graphical elements to different screen resolutions.

In the statement above, all the parameters to the statement are static, meaning they will be calculated only once and stored together with the paintcomponent instance. To give parameters that are dynamic we have to encapsulate them with bracelets { }. If the statement above had the attribute "[. . .] width {$width}", then the statements

```
set width [- $width 10]
flush # refreshes the screen
```

Listing 19.4: Interfacing Dynamic GUI Parameters

would have changed the appearance of the rectangle. The flush statement is the repaint() statement in the scripted client.

The paintscript is executed on init. The Interpreter calls up CodeThing to evaluate, which triggers stanza, which again requests the responsible operator, which in this case is GUIcmdFacade, which points to the GUI class. The GUI class serves as any other operator, with a number of additional methods tailored for GUI components to translate from the script to Java. The method for the current command is located through a switch. The arguments are assembled, and the dynamic arguments are parsed. The arguments for a GUI statement are sent in a properties instance. The Properties class allows for the arguments to have names, and to provide default values for these arguments.

```
# statement that uses the properties class
paintstring value "hello world" x 50 y 50
# statement that do not use the properties class
starttrigger $myTrigger false
```

Listing 19.5: Properties Example

After the argument has been assembled and treated, a paintcomponent instance is created.

PaintComponents

Paintcomponent is a class that collects all the data that is necessary to draw a primitive and compute the dynamic attributes. The GUI class stores all the paintcomponents from a script in a Vector associated with the script, that are fetched on demand from the controller window. The paintcomponents available for the scripted client are:

- line	- rectangle
- animation	- string
- image	setting clip
- setting color	setting font

19.3.2 Animations

To prevent overhead by scripted animations, the scripted framework include a module for Animations. This module contains animation classes for the most common animations, and support for new ones to be added when they are needed. The current implementation contains the animations we decided to implement in Section 12.8.1, which was Stringrotator, Imagerotator and Progressbar. All the animation classes extend an abstract animation class, making interface for the animations easier. Interaction between scripts and animations are done through the AnimationCmds operator. To draw an animation, it requires to be registered as a paintcomponent.

All the animations run in the same thread. When an animation object receives orders to animate, it will register to the Animator class that executes the animation code for each animation registered on it. For each iteration the Animator polls the Animation object to check if it is done animating. When an animation is done, it leaves the Animator thread. The animator is created to reduce the number of parallel threads executing, as this is a weak point on the resource constrained devices. The thread executes as long as there are still animation elements registered on it.

Operations on the animation are done through the AnimationCmds operator. All the animations must support commands for speed, start, stop and reset. Some of the commands are tailored for specific animations.

Animation example

Listing 19.6 show an example of how to use an animation element in the MOOSES client scripting framework. The imagerotator is first declared in the script with positioning and additional data such as the direction to rotate and the anchor options of the graphics component that shall paint it. After the imagerotator has been initialized in the script the speed of the animation is increased and two pictures are added.

The paint tag contains a statement that registers the imagerotator as a paintcomponent, providing for it to be painted when the display repaints. The "! 5" tag says that the rotator should rotate to the next image whenever the user presses the five-key.

```
canvas imagerotatorexample begin{

! main
# create an imagerotator
set imgroter [imagerotator x [- $reswidth 50] y 10 width 40 height 40 direction vertical
    anchor top|hcenter]
# set the speed to animate
speed $imgroter 6
# add the images
addpic $imgroter [image /res/ball.jpg 40 40]
addpic $imgroter [image /res/fly.jpg 40 40]

! paint
animation $imgroter # paint the animation

! 5
presscript:
next $imgrot # display the next image
stop}
```

Listing 19.6: Using an imagerotator in the scripted client

19.3.3 Input

In the previous version of the client we managed user input by identifying the key through a switch, compute the code, and send a keyMsg to the server. The opportunity to send a key message has been maintained in the script, because a key press might be all the game needs to take action. Sending a KeyMsg is done without interacting with the script, and therefore the overhead from the interpreter is removed. In the scripted client we instantiate an object that contains all the data concerning a key.

```java
  private class Input{
        KeyRepeater keyrepeater         = null;  // thread that repeats the key when pressed
        public String pressScript       = "";    // Script that executes when pressed
        public String releaseScript = "";        // Script that executes when released
        public boolean sendPressed      = false; // flag for sending a keyMsg on press
        public boolean sendReleased = false; // flag for sending a keyMsg on release

                // variables used by the keyrepeater
                public boolean repeatkey        = false; // flag indicating whether the key
                    should repeat
        public boolean pressed          = false; // flag keeping the repeater alive
        public int interval             = 1000;  // repeat frequency, millis

                // loads the keyrepeater
        public void loadRepeater(int key){
                keyrepeater = new KeyRepeater(key);
        }
}
```

Listing 19.7: Input class

The Input instance may be written as a tag (to increase readability), or as statements. The properties of an Input may be manipulated runtime. This is convenient because some keys might not have a function before an event has occurred. An event may also alter the way the key is behaving.

To repeat a key that is pressed, the client uses a KeyRepeater that runs in it's own thread. Although the Canvas class has support for repeating keys, we choose to make our own implementation because the one provided by the Canvas class does not support changes in the frequency of repeatings.

The keyrepeater represents a problem for some phones, because every repeater starts its own thread, which might result in multiple threads fighting for the CPU. This problem might be solved by executing all the repeaters in the same thread, and maybe even use the animation thread.

19.3.4 Sound feedback

Sounds are declared in the MOOSES scripting platform by giving the path to the file using the *sound* command. The sound support is based on MIDP 2.0's Media library. Using the *sound* command includes instantiation of a Player for each sound added. The player is pre-fetched and realized, meaning that it loads the entire sound file into the memory.

Sound files are preferably loaded in the ! main part of the application, because it is a relatively resource expensive task. The SoundCmds class provides interface for interacting with the initialized sounds. Basically these commands are just play and stop, where start may have an additional parameter telling the sound to loop.

Example of sound usage

The code provided in Listing 19.8 shows usage of the sound interface. The sound is initialised in the main tag as *gunShot*. If the user presses the fire button or the 5 button the client will send a gunshot message to the server, and play the sound of the gunshot.

```
canvas soundexample begin{
! main
set gunShot [sound "/sounds/gunshot.wav"]
...
! fire
sendscript:
send gunshotMsg
playsound $gunShot

! 5
sameas: fire
stop}
```

Listing 19.8: using sounds in a scripted client

19.3.5 Communication

Data are sent between the game server and the client using the AFPropertyMsg. This message comprises an identifier that is the type of the message, and a hashtable containing the values. The identifier and the hashes has to be equal on both sides (Server and client), so that the actor may identify the message to know which data it possess.

Sending data from the script is done through the command *send*. This command takes the name of the message and name of the hashes, and their value as parameter. Since all values in the script are represented as Strings on the java side, additional parameters is necessary to cast the value to its real type. Otherwise they will be sent as Strings, and may be casted at the server side. To cast a value, an identifier representing the target type has to be placed in front of the certain value.

Communication example

```
send valuetest strval hello intval %i 50 doubletval %d 1.55 boolval %b true longval %l 20000
    objectval %o $myImage
```

Listing 19.9: Sending data to server from script

In the example above, the script will send an AFPropertyMsg called *valuetest*. The argument of the send statement further consist of values to put in the hashtable, with the hash first and the value following. The tokens introduced with the '%' are the casting identifier. The first hash, strval, has no cast identifier because it is supposed to remain as a String on the Java side.

Receiving data is not done through a command interface. The developer has to define tags in the script that are called the same as the identifier of the data message. The tag is followed by the script to execute when the data is received. The controller window contains an interface for receiving the data. It will translate the data to the script, and call the responsible script to execute. The data is available in the script through the $msghash variable that is set by the controller window.

19.3.6 Triggers

One of the first things we wrote for the scripted client was the support for threads. The threads work ok, but they produce a lot of overhead, and might be difficult to control. We figured that a trigger, defined in its own class and not as part of the core, that gave good opportunities for control and manipulation would be more convenient.

The trigger is intended to keep more of the attributes on the Java side, and to execute a pre parsed script. The trigger runs the script either once, or repeatively. The trigger may also contain scripts that executes when the trigger exits the run loop.

The trigger is controlled through an operator TriggerCmds that operates on it. The operator provides commands to start, stop, pause and revoke the trigger. It also provides commands to manipulate the trigger, such as the frequency of repeating, number of repeats and change the scripts.

Trigger example

```
canvas triggerexample begin{
! main
set countdown 10
set myTrigger [trigger interval 1000 periodic true start true itterations 10 code {
incr $countdown -1

flush
} onexit {
set countdown "explode"
flush
} ]

! paint
paintstring value {$countdown} x 50 y [/ $resheight 2]

stop}
```

Listing 19.10: Using triggers in the scripted client

Listing 19.10 shows an example of how to use a trigger. The trigger will draw a string on the screen that counts down from 10, and displays "explode" when it reaches zero.

Part V

Testing

CHAPTER 20

Performance Testing

In this chapter we will test the current implementation of the scripted client to measure and compare performance with the previous client implementation. The programs will be executed through the Sony Ericsson profiler provided with their development kit , and we will reflect the bottlenecks and differences on the two implementations.

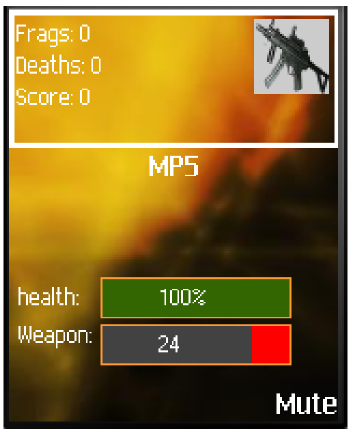
20.1 Profiler Measurements

The tests will allow us to reflect local computations consumption of processing power. Both tests will be applied on simulated versions of the SlagMark clients that do not connect to a server. It is convenient to compare these two applications, because they respond with similar actions to events, and they use the same resources (Images and sound files). Testing will be done five times in all cases, and the average number of instruction cycles will be presented. The tested scenarios follow below.

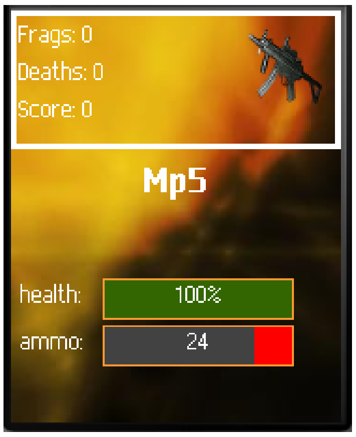
20.1.1 Starting the application

The first test will be a start of the application (the canvas, not the entire client), without performing any actions. The goal of this test is to measure the processing consumption on initiation of the canvases, and compare them.

The profiler shows that the old client used 17.1 million instruction cycles on the init with 90 percent of these used on resizing images. The scripted client used 4.5 million with 87 percent



(a) old client for SlagMark



(b) scripted client for SlagMark

Figure 20.1: Screenshots from both clients

on initiation and parsing of the script.

From these results it might seem that the scripted client is more efficient for initiation than the previous client, however, as stated, 90 percent of these cycles is used on resizing pictures, which is handled differently by the script.

As we can se from this test, there are resources to save on reducing the script initiation process on the client. If we had completely parsed scripts on the server, and sent them to the client, all the client would have to do was to fill in the data structures.

20.1.2 Painting the screen

The next step was to measure the number of instruction cycles used for the client to refresh the screen with the repaint method. The profiler gave us 65.5 thousand instruction cycles with the old client and 218.1 thousand with the new. The scripted client consumes more processing because the paint method on the scripted client include some script executions of pre-parsed code segments and some detours compared to the old client. The detours are a consequence of the scripted client often has to traverse multiple modules to fetch variables.

20.1.3 Performing shot

We will provide another test that involves pressing the same buttons on both the implementations. The first key will be the key that fires the weapon. This is convenient, because both the clients executes code that achieve the same result, but on different places. The old client triggers a method that contains all the logic, where the new client has to execute a pre-parsed script through the interpreter.

The profiler gave us 1328 instruction cycles on the old client against 31.9 thousand on the new scripted client.

The script interpreted on keypress is relatively heavy, meaning that the scripted client executions involve a lot of interpreting.

20.1.4 Changing weapon

Next out we pushed the change weapon key on both devices. Changing weapons on the client involves rotation of the pictures and the string indicating the selected weapon. This will allow us to measure the animation code on the old client against the implementation for animations on the scripted client. The old client came best out with the total of 1859 instruction cycles used on animation logic against the scripted clients 3664 instruction cycles. The scripted version contains some redundant code to make it more generic, but it has only a few calls to the interpreter. In this case the only call is to calculate the string on the string rotator, and the script to trigger the animation.

20.1.5 Sending key message

Pressing a key that does not contain a script should be relatively equal on both clients, however the scripted client has to make a few detours to access code and variables and to check if conditions are met. The profiler gave us 170 instruction cycles from the old client and 737 instruction cycles from the new. The codes for releasing buttons are almost the same. The profiler gave us 49 instruction cycles from the old client against 631 from the new client.

20.1.6 Thread and triggers

In the last test we measured the cycles used by the mechanisms that reload the weapon on the client. This is done using a thread on the old client and through a trigger that executes parts of the script on the new client. The thread on the old client used 8606 instruction cycles in total against the 240.8 thousand instruction cycles used by the trigger on the scripted client.

20.1.7 Comparison

In this section we will compare the profiler results from both clients to measure the performance we lose by using the new implementation. In this comparison we will also present the time of execution on a 200 MHz CPU which is about the lowest threshold you could expect from modern mobile phones. Table 20.1 shows the profiler results measured in instruction cycles needed to perform the operation and the factor New client / Old Client to indicate the difference.

The table also presents the time the execution of the instruction cycles using a 200 MHz CPU, and the difference the difference in time, new client - old client.

	Old client		New client		Difference	
Test	Instruction Cycles	Execution Time	Instruction Cycles	Execution Time	New/Old	Exec. Time
Initiation	18660683	0.0933	5801677	0.0221	0.31	0.0712
Painting	65503	$3.275 * 10^{-4}$	218119	$1.09 * 10^{-3}$	3.33	$7.625 * 10^{-4}$
Fire weapon	1328	$6.64 * 10^{-6}$	32926	$1.6463 * 10^{-4}$	24.79	$1.578 * 10^{-4}$
Change weapon	1859	$9.295 * 10^{-6}$	3664	$1.832 * 10^{-5}$	1.97	$9.025 * 10^{-6}$
Send key	170	$8.5 * 10^{-7}$	737	$3.685 * 10^{-6}$	4.34	$3.44 * 10^{-6}$
Release key	49	$2.45 * 10^{-7}$	631	$3.155 * 10^{-6}$	12.88	$2.91 * 10^{-6}$
Execute Thread	8606	$4.303 * 10^{-5}$	240777	$1.2 * 10^{-3}$	27.98	$1.157 * 10^{-3}$

Table 20.1: Profiler results

As we can see in Table 20.1 the scripted implementation produces a significant overhead, especially with respect the scenarios that involves execution of larger scripts. The results presented in this chapter are done on a client that is optimized considerably since the test session at Kosmorama.

CHAPTER 21

Testing of requirements and quality attributes

This chapter will reflect to what extent the scripted client implementation covers the functional requirements and the quality attributes set in the architecture.

21.1 Functional Requirements

This section will reflect testing on the functional requirements.

FR1: The implementation must support graphical components	
Executor	Sverre Morka
Date	18.05.2007
Time used	2 hours
Evaluation	Success: The basic primitives supported by Java Graphics is supported

FR2: The implementation must support dynamic changes in graphical components	
Executor	Sverre Morka
Date	18.05.2007
Time used	2 hours
Evaluation	Success: By defining parts of the argument as dynamic, they may be changed during runtime

FR3: The implementation must support sound feedback	
Executor	Sverre Morka
Date	18.05.2007
Time used	2 hours
Evaluation	Success: Sounds may be instantiated and played on demand by the script

FR4: The implementation must support logic for user input	
Executor	Sverre Morka
Date	18.05.2007
Time used	2 hours
Evaluation	Success: The script supports logic for user input by pressing, holding and releasing keys.

FR5: The implementation must have support for the most common animations	
Executor	Sverre Morka
Date	18.05.2007
Time used	2 hours
Evaluation	Success: The script supports instantiation and manipulation of the animations found in the profile section, although there are more to be added to a complete framework.

FR6: The implementation must support multiple canvases	
Executor	Sverre Morka
Date	18.05.2007
Time used	2 hours
Evaluation	Success: Script is defined as multiple canvases that constitute the client application for a certain game

FR 7: The implementation must support interface for communication from and to the server	
Executor	Sverre Morka
Date	18.05.2007
Time used	2 hours
Evaluation	Success: Communication to the server is done by using the send command. Tags are written to respond to data from the server

FR 8: The implementation must support interface for independent logic such as threads	
Executor	Sverre Morka
Date	18.05.2007
Time used	2 hours
Evaluation	Success: The triggers provided in the framework for scripting clients runs independent logic. The triggers are controlled from the script.

21.2 Reflection on quality requirements

In this section we will reflect to what extent the quality requirements have been met.

21.2.1 Availability

This section reflects the test results of the availability requirements.

A1 - Packet loss	
Executor	Sverre Morka
Date	20.05.2007
Stimuli	The server sends a status message to the client
Expected response	The Server synchronizes with the server once a minute
Observed response	The client is synchronized by update messages sent from the server, once a minute if the client has not received update within the last minute
Evaluation	Success: During several hours of testing, there was no problem with the client getting out of sync.

21.2.2 Modifiability

This section reflects the test results of the modifiability requirements.

M1 - Extend the interpreter with additional commands	
Executor:	Sverre Morka
Date	03.03.2007
Stimuli	Extend the command vocabulary of the interpreter
Expected response	The framework developer provides the additional Cmds- and corresponding classes to the client Jar file, and makes the interpreter load the new modules without modifying other parts of the client.
Observed response	If the new commands do not interfere with the functionality from the underlying canvas, only manipulation of Cmds- and additional classes are needed.
Evaluation	Success: Adding new commands through cmd classes will make them be loaded and ready to be used by the interpreter.

M2 - Change dynamic game related data	
Executor:	Sverre Morka
Date	20.05.2007
Stimuli	The user's actions make changes in the local variables
Expected Response	The controller canvas runs the script associated with the action
Observed response	The controller canvas runs the script associated with the action, and may send data and/or manipulate local data controlled by the interpreter.
Evaluation	Success: Game logic is provided by smaller script fragments that are executed on actions such as user input, triggers or data exchange.

M3 - Extend the framework with more states	
Executor	Sverre Morka
Date	03.03.2007
Stimulus	The framework developer wants to extend the domain of the framework to fit other concepts.
Expected response	Manipulation of the backbone state machine should be possible without side effects
Observed Response	As expected
Evaluation	Success: During the development of the scripted client we made some changes to the states. Changes did not affect the rest of the client.

M4 - Adding new games to the client	
Executor:	Sverre Morka
Date	18.05.2007
Stimulus	The maintainer wants to add more games on an implementation of the concept.
Expected response	The Maintainer adds the game to the server together with the corresponding script and client resources, without interfering with any code or modules
Observed response	The script is loaded dynamically, but additional resources has to be stored on the phone in advance
Evaluation:	Partly success: The ability to dynamically load resources on the client is available from ServiceFrame, but is not installed on MOOSES yet. This requires some manipulation of the server as well, it will be installed at a later stage. However, addition of new games does not interfere with the source code

21.2.3 Performance

This section reflects the test results of the performance requirements.

P1 - Response time from player actions	
Executor	Sverre Morka
Date	20.05.2007
Stimulus	The user gives input to the game through his/her controller (mobile)
Expected response	The action chosen by the user should play out on the screen(s) in less than 0.5 seconds.
Observed response	The action chosen by the user plays out on the screen(s) in less than 0.5 seconds
Evaluation	Success: The scripts executed on user input may manipulate dynamic data from the paintcommponents and call a repaint. Phones with limited processing power may suffer latency in the display update.

P2 - Fair premises	
Executor	Sverre Morka
Date	20.05.2007
Stimulus	The client runs on a resource constrained device
Expected response	The gameplay experience should not be affected by the limitations of the client
Observed Response	The resource limited devices tested on the former client suffered some latency, and minor bugs
Evaluation	Fail: This requirement was tested on the Kosmorama presentation with a more resource consuming version of the scripted client. Tests with the optimized version however has proved to run on the clients tested on both implementations without noticeable latency.

21.2.4 Security

This section reflects the test results of the security requirements.

S1 - A player tries to cheat	
Exectutor:	Sverre Morka
Date:	18.05.2007
Stimulus	The user tries to alter the script that represents the game controller
Expected response	The client downloads a new script whenever a game is started, or a user logs on.
Observed response	At this point, the client only receives the script if it is not already stored on the mobile in advance
Evaluation	Fail: The server will not send a script to the client if a script with the same name is already present in the RMS.

21.2.5 Testability

This section reflects the test results of the testability requirements.

T1 - status of running modules	
Executor:	. . .
Date:	. . .
Stimulus	The system maintainer wants to check the status on the running system.
Expected response	The framework must supply a GUI based tool for debugging of the system. This GUI must also profile attributes related to the scripted client, and the game running on the game server
Observed Response	A development tool for development and debugging has not been made yet.
Evaluation	Fail: Not implemented

T2 - Test a new game-client script	
Executor:	...
Date:	...
Stimulus	The game developers want to test the controller they have scripted for their game.
Expected Response	The Framework must include tools for writing and testing scripted controllers. Such a tool should also be able to profile the program and display the measurements in wanted format.
Observed Response	...
Evaluation	Fail: No editor has been made yet

T3 - Game developers wants to test their client	
Executor:	...
Date:	...
Stimulus	The game developers test their client without having to run the game
Expected response	The framework must simulate server connection, and provide an interface for simulations.
Observed response	Simulation of the server is hard coded during test time, no interface for this in an editor.
Response Measure	Fail: No development kit has been made for the development yet

21.2.6 Usability

This section reflects the test results of the usability requirements.

U1 - The user wants to exit the application	
Executor:	Sverre Morka
Date:	18.05.2007
Stimulus	The user wants to exit the application
Expected response	The procedure for quitting the application is the same in every part of the application, independent of the scripts
Observed response	The Exit procedure is easy accessible from all the application states, but is customized by the script author
Evaluation	Partly success: The player may exit in all states of the application, although it might not be the same procedure from all game clients. Also, this requires the client developer to write exit support.

U2 - The user wants to log in to the system	
Executor:	Sverre Morka
Date:	20.05.2007
Stimulus	The user wants to log in to participate in the games
Expected response	As long as the mobile satisfy the hardware requirements and the server is not full, this should be accomplished every time a player tries to log on
Observed response	The client sometimes hang on login screen, sometimes it takes a very long time, sometimes the phone is not able to initiate the script
Evaluation	Fail: The client is logged on to the system in 4 of 5 tries, but there are still bugs that have to be dealt with.

CHAPTER 22

Test-session at Kosmorama with real users

Our project was participating in the Kosmorama film festival [1] and we decided to make a scripted version of the client ready to use it for test sessions at this festival. At this point we had two complete test-games that needed a client. Morten Versvik had made a new game called BandHero as a compliment to the popular game Guitar hero. This game requires minimal logic executed on the client, and so the client was written on five lines of code. A presentation of the BandHero game can be found in Appendix A.

In addition to the client for Bandhero, we had to make the client for the game we used in the previous test sessions. At this point we had already made solutions in our version of Hecl to support for the elements required by this game. Most of the code was also provided since we had used this as a benchmark to test the GUI during development. We wrote the code for input and communication and ended up with a script with approximately 200 lines (the script is provided in Appendix A).

The script worked fine through the simulator, but suffered a few bugs when distributed to the phones. The phones we used to test the scripted implementation were Sony Ericssons K750, which had proved to run our first implementation without problems related to latency. We also tested the script with the newer K- model from Sony Ericsson, namely K800i, which did not suffer from the latency problems. This meant that we had to optimize the script solution further. In this context it is worth to mention that we have tested the optimized version used in Chapter 20.1 on the K750 phone, and it gave no visual differences compared to the K800i.

22.1 Reducing latency

Our solution for repeating keys was not synchronized, resulting in it to spin several threads for each key pressed. We made a new version which synchronized the thread, so that there would only be one thread running for each key to press. This worked as a temporarily solution because the clients we had at this point only required one key to repeat. However, for further development, we decided to cut down on the potential threads running to just one for all keys that had to be repeated.

The K750 did not manage to repaint the canvas when the user provided input too frequent. The framework has an overload of repaint calls in the first place, and with the script implementation, a call to repaint is a bit more time consuming task. The first thing we did to cut down on this was to parse the dynamic fields of a PaintComponent Object before it was being used, meaning that it were only parsed one time in stead of each time it were used. We also removed some of the calls to repaint in the GUI components provided by ServiceFrame. The problems with the mobile not being able to refresh the screen was reduced, but not eliminated.

Although some of the overhead was eliminated, it still produced much more overhead than the current scripted solution. The version that is provided with this report has been developed further, pre-parsing all the scripts used by the application, not only the draw scripts.

22.2 Test session

We tested our concept at the Kosmorama festival on two occasions. The first demo was done for the Norwegian television program Newton, followed by an open test session were everyone could join in. This test session provided us with information we had speculated in for since we started the MOOSES development. This was the first time we got to test with more than four players, and we had thirteen phones connected to the system at the max. Testing at Kosmorama also gave us the opportunity to test the client on more phones than we had done earlier. The phones we used at Kosmorama were all Sony Ericssons phones, the models were W850i, K610i, K800i, K750i and W300i. Technical information on these phones can be found at the Sony Ericssons product guide [17].

The first three phones worked fine with the scripted solution, but the K750 and the W300i encountered some latency both for the drawing and the communication. These phones had problems running more threads resulting in a lot of lag, and W300i had the smallest display of all the phones used, 128x160 pixels, which gave a disadvantage because some of the elements did not fit the screen (Text Strings went outside the display).

As expected we encountered some bugs during the tests. Most of these bugs were related to the server and the game, but some were related to the scripted client.

- Sometimes the KeyRepeater hung
  On some occasions the worm continued to fire although the button had been released.

This was probably due to the script being interrupted and did not get enough time to execute on the resource constrained phones.

➢ Sometimes the player is able to shoot even though he is dead
This one is odd, because when a player dies, the client will switch to another canvas. Therefore the communication for the main game canvas should not be enabled. This error is limited to about a second after the death incurred. This problem is probably due to communication latency, because the player is dead in the game before the client gets notified. This error might be prevented by modifications to the game rather than the client.

➢ Sometimes the phone is not able to initiate the script
This error occurred a few times. It is likely but not certain that the reason is that the phones were not able to initiate the script. This error occurred after the voting session, the vote animation stopped, indicating that the client had gotten the message to start the game. At this point, the client freezed. However, after a restart of the client application, the mobiles were able to initiate the script. It is also possible that the animation used by the voting canvas use too much resources on the constrained devices, allowing less memory heap to be used to initiate the scripts.

➢ One time the client hung
The client hung, and the mobile did not respond to anything. This resulted in having to take out the battery for restart.

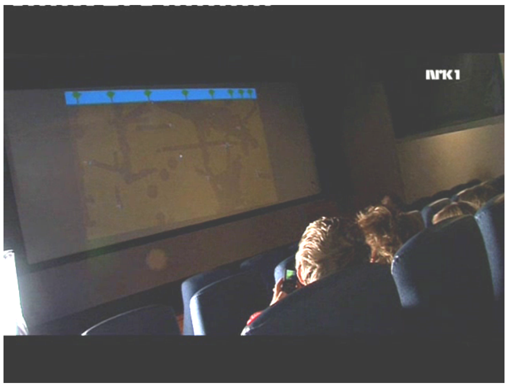
➢ Movements hangs up, no communication
Sometimes the worm on the game did not respond to the keys being pressed on the client. However, the actions were remembered later, and then executed in the game, resulting in a latency ranging from one to five seconds. Communication and processing limitations is assumed to be the main reason to this problem.

The problems mentioned above mainly occurred on the K750 and the W300, but latency were also a problem on the more powerful phones. The profiler results presented in Section 20.1.7 is based on a client that is about four times more efficient than the client used in the test session at Kosmorama, and it is therefore likely that the resource constrained phones will have less trouble running the client.

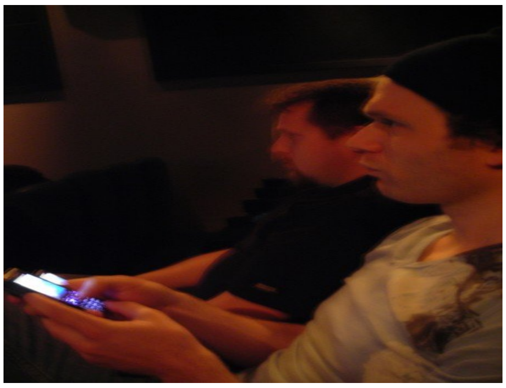
As we were expecting, the largest issue showed to be latency from the communication. This was the first time we tested this amount of clients at the same time, and the delay might be explained as a result of interference and much processing by the Bluetooth router and the server. The sizes of the messages used by the scripted client are larger than they were the last time we tested the concept at Nova, and we can reduce the latency by reducing the size needed by the data carriers. An optimized version of the client has been made with the keypress message small enough to fit one Bluetooth package, but we have not been able to test it with more than two clients. The AFP message that contains additional data will also be replaced by a new optimized message based on Java Vectors in stead of hashtables.

22.2.1 Images from test session at Kosmorama 2007

(a) Morten explains the MOOSES concept



(b) In-game screenshot


(c) Happy testers

(d) In-game screenshot

(e) Audience

(f) In-game screenshot

Part VI

Discussion and Conclusion

CHAPTER 23

Discussion

In this chapter we will try to discuss the things we have learned during this master thesis, and tests of the prototype to be able to answer the research questions.

23.1 Flexibility

One of the goals of this project was to find out whether or not we could achieve a more dynamic client for MOOSES using a scripting platform for game clients. During this project we have been able to build and test a prototype as proof of concept for use of scripts to define the game clients.

We have also tested sending the script from the server when starting games. However, at the present point there is no support on the server to provide the rest of the resources, meaning that these have to be in the phones RMS in advance. ServiceFrame has support for sending resources, and we will have to implement this feature to the scripted client to make the MOOSES client completely independent of pre-allocated resources. Using ServiceFrame for resource management will also allow us to perform heavy operations on the resources, like resizing a picture, on the server. Implementation of resource management from the server will not change the interface from the script because the state machine will request these resources automatically if they are not located in the mobiles RMS. This will result in some latency the first time resources are loaded.

With the implementation of dynamic resource management on the server we will be able to provide for MOOSES to be dynamic. This means that a MOOSES implementation will support adding and removing just by adding the game files, the client script and the client resources.

23.2 Efficiency

As we can se in Section 20.1.7 the scripted client consume up to 28 times as much CPU resources as the former client by executing similar tasks. More advanced clients may increase this factor even more.

23.2.1 Communication

The prototype also lack efficiency on the communication especially with respect to the AFPropertyMsg that makes use of a hash table. The hashes of the hash table represents data load that can be removed, providing for the messages to be smaller. We could replace the hash table with a Vector to get rid of the extra baggage of the hash table. This will impact the scripting module, making sequence of data in the array important in stead of calling data by their names (hashes).

In future versions of MOOSES we might be able to use wireless fidelity (WiFi) to communicate with server, as phones with WiFi support becomes available. WiFi will provide faster transmission rate than Bluetooth 2.0.

23.2.2 Parsing overhead

Hecl is designed to both parse and interpret the script on the mobile device. This way scripts may be edited on the mobile device before they are executed. Our scripted client solution will not need to parse the scripts because it is not intended to manipulate the scripts runtime. Therefore our client would only need to have an interpreter, and be provided with the parse tree from the server. This way, all the latency regarding parsing would be illuminated.

This is basically what is done with the prototype because all the script fragments are parsed in the init. However, the interpreter always checks if the script is parsed before it executes it. And in some cases, this check is a bit time and process consuming.

23.2.3 Detours

Another feature that is responsible for causing overhead is the detours of references. Detours are a consequence of making the modules generic and increase of modularity. In the scripted client implementation the interpreter often has to traverse multiple modules to get to a variable. For instance, it might have to find out where the variable is located, and fetch it from a hash table. Then it has to check whether the variable should be substituted or not. This represents more computing than just fetching a variable from the memory.

23.2.4 Size

The new client comprises more classes than the former client because the scripting framework comes in addition to the other classes. This mainly affects time to download the client from the server. Storage capacity on modern phones is limited to between 30 and 100 Megabytes, and even more when phones have memory cards inserted. Therefore the increase in size is not a major concern for hardware requirements.

23.2.5 Optimizing

There are many points regarding the prototype that shows room for improvements. If we remove the entire parse functionality from the client and leaves this part to development, yielding only the parse three for the client, the script overhead would be limited to just detours. We could also cut down the size of the framework by removing redundant functionality.

23.2.6 Hardware requirements

During our test session at Nova we tested several phones. Some of the resource constrained devices experienced problems that resulted in latency, which again affected the gameplay experience. We had tested some of these phones with the former client implementation without noticeable latency. Unfortunately, we have not been able to test all of the phones used on the Kosmorama test session on the optimized version of the scripted client.

The Sony Ericssons models that we had available seemed to work fine with the optimized version, and we did not experience any notable latency. Nokia phones on the other hand still have an issue running the MOOSES client. We have tested the framework with the Nokia N73, which is one of the newer Nokia phones on the marked. Nokia N73 had some performance problems, sometimes causing the phone to freeze for shorter periods of time. Nokia N73 has a smaller memory heap than the Sony Ericssons mobiles, and this is why we experience these problems.

Using scripts extend the need of memory heap, because it involves more data to compute. During our depth study we also experienced problems getting Nokia phones to run the client. It is likely that memory heap will be extended on future cell phones developed by Nokia, but reduction of heap consumption is nonetheless a prioritized issue.

When looking on the results from performance testing it is obvious that the hardware requirements has been increased, and phones with faster CPU's will gain advantage of being able to compute code faster. The hardware requirements for MOOSES clients were high also with the first implementation. If common phones are able to run the MOOSES client without notable latency at this point, future phones will also be able to do so.

23.3 MOOSES Limitations

In this section we will discuss to which extent the MOOSES platform suffer limitations as a consequence of using a scripting middleware on the mobile client. We have seen that the scripting solution increase the hardware requirements. However, we have been able to test the scripted solution with relatively old Sony Ericssons phones without notable delays. The latency caused by the processing overhead is therefore not a major concern, although the overhead might become noticeable when more advanced scripted clients are developed.

Using mobile phones as clients is a good idea that makes it easier to support multiple players without having to reach to expensive alternatives to provide controllers, but it comes with a drawback. When the mobiles run out of power, they are not able to play anymore. The increase of processing also increases the power consumption, which causes a problem for the MOOSES concept.

23.4 Game development

In this section we will discuss to what extent the scripting platform will make it easier or harder to make game clients. Using a scripting platform will increase the abstraction level, providing for the client to be written on less code. As the abstraction level increases it is also easier to make tools for code generation.

On the other hand, the syntax of our scripted solution takes some time to get into. But the learning curve of our scripting language is lower than the learning curve of the system language Java. Our scripting platform represents a language that is more domain specific, with commands limited to develop game clients for MOOSES. For a game client developer, it will probably be easier to relate to a domain specific language than a language that contains much redundant functionality.

At this point, the scripting platform will make it harder to make clients because of the lack of error correction. Development of games on the scripting platform may be done through any text editor, and the author is responsible for finding the errors himself. Fortunately Hecl supports exception handling that may ease the debugging to some extent.

23.4.1 Game development limitations

The scripting platform puts some constraints on the game client development for MOOSES. Developers have to use the Script language to develop games which put limitations on freedom of creativity. This is because the developers are relatively tied to use the components and functionality provided by the scripting platform, which means that the developers of the scripting platform must foresee the demanded components to be able to provide them. If the game client developer wants to provide new components to renew gameplay he might be able to code these from scratch using fine grained script commands, but this is likely to affect the performance of the client. In Appendix A we can view the source code of SlagMark, which shows that the progress bar in this game was coded by using primitives rather than the one provided as precompiled components by the framework.

Another feature that will be reduced or illuminated by a scripting approach is the ability to move parts of the game to the client. In our depth study we looked on opportunities that provided most of the game on the client and parts of it on the projected canvas. As mentioned in Section 12.5 a problem with racing games is that the vehicle has to be in the display at every time. If we want to build a racer that scrolls over the track, just focusing on the leading vehicles, the other vehicles won't be able to navigate at all. If we could make the client able to display the car and track on the mobile phones display, the players that where not visible on the game canvas would be able to use the display on their mobile phones to navigate.

In Section 12.8.2 we discussed using MIDP2.0's game API. This API increase the abstraction level for game development, making games require less code. This API comprises functionality for Animations, collision detection, sprites, layers and management of layers. We might be able to provide advanced gameplay on the client by interfacing this API from the scripting platform.

Modern mobile phones come with their own Graphical Processing Unit (GPU) for handling the graphical representation. In our depth study we also discussed the opportunity to take advantage of the mobiles GPUs to make games clients in 3D for the MOOSES platform. For instance in use of advanced gameplays, we could make first person shooters on the clients and use the projected canvas to give an overview and zoom down on gunfights etc. 3D games for mobiles are getting increasingly advanced, and this is definitely a feature that we would like to implement for the MOOSES concept. However, this feature will probably be too heavy to develop through scripts.

CHAPTER 24

Conclusion

In Chapter 3 we stated two research questions. We wanted to research on alternative ways to implement dynamic loading of code in Java to make a dynamic client for the MOOSES platform. We also wanted to look on which elements such a solution had to provide to be convenient for client development. The two main focuses on this master thesis have been in which way the client for the MOOSES platform will be affected by a scripting platform with respect to performance and game controller development.

Throughout this project we have looked into central concepts of a scripting solution for mobile devices to work on top of the J2ME and ServiceFrame, and we have developed a prototype.

In this chapter we summarize the projects results by answering the research questions.

RQ-I: **How will an alternative solution affect MOOSES?**

> ➢ During this project we have developed a prototype that is based on an open scripting platform for J2ME enabled devices. The prototype is based on the open scripting platform Hecl which executes both the parsing and the interpreting on the device. The prototype has been a good proof of concept and we intend to base future clients on this prototype. For future scripted clients, the parse process should be executed during development, yielding only the parse tree to the client.

a) Will a scripting API for the client make the MOOSES able to dynamically load new games?

> ➢ The client is able to load the scripts from the server when a game is started. As soon as dynamic resource distribution is

implemented on the MOOSES server it will be able to load the games dynamically without changes in the source code. This means that games can be added without interfering with the source code on the client or the server.

b) To what extent will the performance of MOOSES be affected by a scripted client?

  ➢ The profiler results in Section 20.1.7 show that the scripted client uses up to 28 times as much processing as the old client on some tasks. For tasks that are time sensitive, this will increase the hardware requirements for the mobile phones. The implementation of scripted clients is also more memory consuming which is also a factor that increases the hardware requirements. The communication also suffers more latency with the current scripting implementation compared to the old client implementation. The scripting approach has to be generic, which causes a lot of redundant computation. The Hecl architecture also contributes with overhead, because it checks whether code is parsed or not.

c) Will the MOOSES platform be constrained by a scripted client?

  ➢ The larges issue that comes alive using a scripting platform is the increase in power consumption used by the clients. Increased hardware requirements and restrictions to client gameplay are also relevant issues, but since we have been able to run the client without noticeable latency with cell phones used on both the old and the new client implementation, the hardware requirements is not a major concern.

RQ-II: **Which impacts will a scripting platform have on game development?**

➢ During this master thesis we have developed three game controllers for MOOSES games. When we replace the API for game controller development it changes the way developers has to think to develop controllers for MOOSES games.

a) Will a scripting platform make it easier to develop a game client?

  ➢ At this point, the scripting platform makes it harder to develop games because of the lack of a development tool. However, the scripting platform gives an abstraction level that eases the learning curve and makes the language more tailored to the concept. The abstraction level also makes it easier to build development tools that can generate code and provide error correction and hints, and simulate the application. A complete scripting platform, together with a development tool to use it, will ease the development process.

b) Will a scripting platform constrain the game clients with respect to game play?

  - ➢ Using scripts to develop game client will affect the developers' freedom of creativity because they have to use predefined components in order to maintain performance on the clients. The scripting platform does not provide support for advanced gameplay, or support for 3D games, on the client.

c) Which elements would a scripting API need for game client development?

  - ➢ The elements required for MOOSES clients depends on how advanced the client gameplay should be. The games developed at this point, and nearest future, will have the client work more as a control than part of the gameplay. In Chapter 11 we gave the artefacts that a game client for MOOSES should include, and in Chapter 19 we saw how these artefacts were implemented in the scripted client. Section 12.8.1 presents the elements that should be available for visual representation in a scripting platform for game client development. We have also discussed the opportunities to include control of more advanced gameplay components. To summarize we can put the elements in a list.
    - Reactions to user input
    - Graphical representations of Images, primitives and animations
    - Support for Threads/logic
    - Communication from and to the server
    - Reactions to data received
    - Support for sound feedback
    - Support for multiple screens/canvases and communication between them

    A scripting platform for game development has to include generic components that support these fields, and opportunities to interface them from the script.

To determine whether or not we are going make use of the client scripting platform we have to measure the qualities we are loosing against the qualities we are gaining. The scripted client solution makes the client less efficient, it increases the hardware requirements for the mobile phones, it increases the power consumption on the client, it makes limitations on how advanced the game clients can be, and it constrains the freedom of creativity for the developers.

Using the scripting implementation we are able to make a MOOSES implementation that is totally independent of source code manipulation when adding and removing new games. The scripting platform also represents a language that is more tailored to the concept and makes game development easier. With the advantage of being familiar with the scripting language we were able to develop the game client for SelfFish (presented in Appendix A) in about three hours.

MOOSES has proved to be best suited with games that provide a simple gameplay, and therefore the scripted client solution is sufficient for games developed to MOOSES.

Although we had some problems during the test session at Kosmorama, we have been able to optimize the client to give a more satisfying level of latency.

The scripted client has proven to work on MOOSES as a replacement for the old statically coded client. We will use this client in future versions of MOOSES, although there are still work to be done on the client to make it a complete satisfactory. This work is presented in Chapter 25.

CHAPTER 25

Further work

The client is an important part of the MOOSES framework, and it is crucial that it is optimized, stable, and attractive for developers and yielding as good support as possible for good gameplays. During this research we have researched elements that are needed by a client and solutions to develop game clients for MOOSES. The prototype developed during this project has proved to be a good compliment to the former Java development platform in context of game development. There is much work left to be done to provide a complete attractive and reliable development platform for the MOOSES games.

25.1 Small scale

These are elements that are relatively easy to implement for the framework without consuming too much time and research.

➢ **Dynamic loading of resources** - At the end of this master thesis, the prototype implementation is able to fetch the script from the server but it still rely on the resources in form of pictures and sound files already being stored with the client. ServiceFrame has support for dynamic loading of resources. Using this functionality is convenient because image resizing, which is the most process consuming task done by the client, can be performed at the server.

➢ **Optimizing the communication** - The current implementation uses a hashtable as carrier for data exchanged between the server and the client. Even though the data carried may be small, the hashkeys extend the size of the message, resulting in a great number of packages being sent over Bluetooth. The future implementation will use a vector in favour of the hashtable. The size of the keyMessage will also be reduced to fit one Bluetooth package. Implementation of new communication protocols should only take a few hours in addition to a few hours with testing.

- **Animations and Graphical Elements** - The scripted client prototype does not support all the graphical elements found in Chapter 12 yet. These, in addition to potential others that we have overseen, has to be implemented to the framework.

- **Accommodation of the server** - The server has to be manipulated to be able to prepare the resources used by the client, and send them over. The current prototype could also benefit from scripts being parsed on the server or during development. The server also has to be issued funcionality to distribute the actor address of the player's on the same team, in order to provide communication between the players for cooperative games.

25.2 Large scale

These elements require some additional research and time for development.

- **Client Optimizing** - Although we have optimized the client several times during this project, it is possible to optimize it even more. As we saw in Section 20.1.7 the scripted implementation produces a pretty big overhead. There is a conflict of interest between Hecl and our purpose. Hecl is not intended to replace Java, but to compliment it. Our implementation is indented to replace Java. By leaving all the script parsing during development, and storing all the information in parse trees that are executed when the client is executed we can illuminate much of the overhead. It might be more convenient to build a domain specific solution from scratch, rather than tailoring a general purpose scripting platform like Hecl. Eluminating the parse process from the client will also reduce the memory heap consumption on the mobile phones.

- **Development Platform** - Game development on the scripted platform is done through text editors, with no support for highlighting of keywords or error feedback. To make the development platform more attractive for game developers it would be convenient to have an editor tool that provided these features together with support for script generation and compilation of the scripts into permanent parse trees (Hecl Stanzas).

- **Code Syntax** - There are, at this point, two people that has developed game client using scripts. The syntax of the scripts that constitute the game clients is based on the syntax that the authors of the scripting platform found intuitive. It would be appropriate to make some research and testing to find the learning curve of the language, and the points that could be changed to make a more intuitive language.

- **Implementation of MIDP 2.0 Game API** - As mentioned in Section 12.8.2 we could take advantage of an interface for using MIDP 2.0 Game API elements through the script. This would require research on how to make games in this API, and how this could be accessed and developed through a script. It would also require some changes to the scripted game client implementation which is based on MIDP Canvas, and not GameCanvas which this API builds on.

Part VII

Appendix

APPENDIX A

Scripted Game Clients

During this project we have made three scripted clients for MOOSES games. The first scripted client was based on the game we developed to test our frame work during our depth study. Morten Versvik developed two additional games during his master thesis, and we have developed a client for these based on the scripting platform.

A.1 SlagMark

SlagMark is the first game we made for MOOSES. The game is based on the old game Liero, and has inherited its gameplay. The game is set to a battlefield in 2D where each player is in control of one worm. The players have an arsenal with five weapons they can use. These weapons are pistol, machine gun, grenades, bazooka and a nuclear bomb. The goal of the game is to be the worm with most kills (frags). The game is instant action, and run in a window of five minutes.

When the five minutes have passed, the highscores is displayed on the game canvas.

A.1.1 Gameplay

The game starts with all the worms placed in random locations on the map. The worms have to shoot their way through the dirt that constitutes the battlefield, in their chase for another player to frag. When a player dies, he get the name of the worm that frag'ed him and have to wait ten seconds to the next insertion.

The controls comprise a jetpack button that allows the worm to jump or fly, horizontal movement, aiming up and down, changing weapon and fire weapon.

Ammunition is limited on the different weapons, and when the player has used his magazine of ammunition he has to reload his weapon. The different weapons use different times to reload, depending on how powerful they are. The Nuclear bomb will take about a minute to load.

A.1.2 Goal of the game

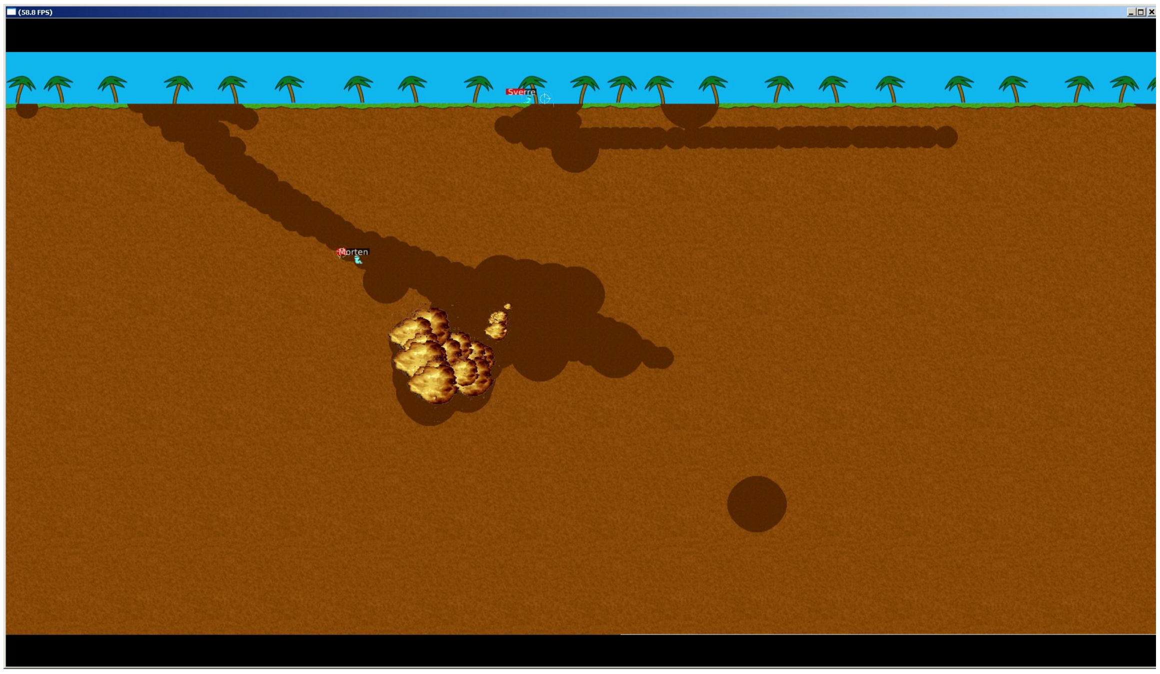
The game is based on total annihilation of opponents and landscape. The players score points for frags, and loose points on suicides. The goal of the game is to achieve the highest score.

A.1.3 Client source code

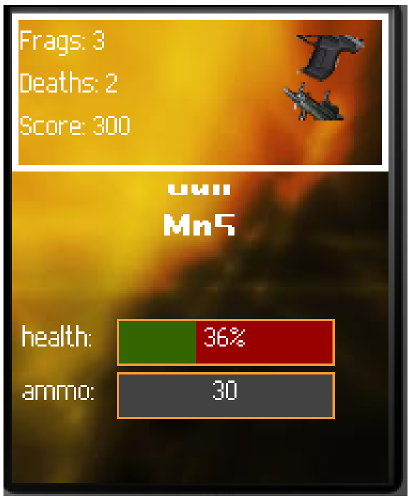
```
canvas maincontrol begin{

! main
set frags 0
set deaths 0
set score 0
set health 100
set ammo 100

set maxammo [list 13 30 4 8 1 ]
set reload  [list false false false false true]
set ammostatus [list 13 30 4 8 0 ]
set repeatkeys [list true true false false false]
set repeatint [list 300 50 0 0 0]
set paintammo [list 100 100 100 100 0]

set weaponpointer 0
set max [lget $maxammo $weaponpointer]
set curr [lget $ammostatus $weaponpointer]

set reloadinterval [list 30 30 200 500 800]

keyrep 5 [lget $repeatkeys $weaponpointer] [lget $repeatint $weaponpointer]


set killername ""

sendpressrelease 2 4 6 8 * left right down up

set reloader [trigger interval 200 periodic true itterations 2 code {
incr $ammo 2
set curr [/ [* $max $ammo] 100]
flush

} onexit {
set ammo 100
set curr [lget $maxammo $weaponpointer]
lset $reload $weaponpointer false
} ]

set sounds [hash {}]
hset $sounds 0 [sound /Sounds/m82.wav ]
hset $sounds 1 [sound /Sounds/m16.wav ]
hset $sounds 2 [sound /Sounds/Grenade.wav ]
hset $sounds 3 [sound /Sounds/RocketAway.wav ]
hset $sounds 4 [sound /Sounds/Gun.wav ]

set background [image "/icons/Background.jpg" $reswidth $resheight]

set img1 [image "/icons/pistol.png" 40 40]
set img2 [image "/icons/MachineGun.png" 40 40]
set img3 [image "/icons/Grenade.png" 40 40]
set img4 [image "/icons/bazooka.png" 40 40]
set img5 [image "/icons/Nuke2.gif" 40 40]

set imgroter [imagerotator x [- $reswidth 50] y 10 width 40 height 40 direction vertical
    anchor top|hcenter]
set strroter [stringrotator x [/ $reswidth  2] y 80 direction vertical]

speed $imgroter 6
speed $strroter 4

addpic $imgroter $img1
addpic $imgroter $img2
addpic $imgroter $img3
addpic $imgroter $img4
addpic $imgroter $img5

addstring $strroter "Gun"
addstring $strroter "Mp5"
addstring $strroter "Grenade"
addstring $strroter "Bazooka"
addstring $strroter "Nuke"

strrotfont $strroter size large style bold
strrotanchor $strroter top|hcenter

proc fireweapon {} {

if { [lget $reload $weaponpointer] } {
return
} else {
        sendshoot
        incr $curr -1
        set ammo [percent $curr $max]
        playsound [hget $sounds $weaponpointer]

                if { [<= $curr 0] } {
                        lset $reload $weaponpointer true
                        triggeritterations $reloader 50
                        triggerinterval $reloader [lget $reloadinterval $weaponpointer]
                        starttrigger $reloader
                }
        }

        flush
}

proc sendshoot {} {
send ShootMsg weapon "%i" $weaponpointer
}

proc incrpointer {} {
incr $weaponpointer
if { [>= $weaponpointer [llen $maxammo]] } { set weaponpointer 0  }
}

proc changeweapon {} {
stoptrigger $reloader true

lset $ammostatus $weaponpointer [+ $curr 0]
lset $paintammo $weaponpointer [+ $ammo 0]
incrpointer
keyrep 5 [lget $repeatkeys $weaponpointer] [lget $repeatint $weaponpointer]
set curr [lget $ammostatus $weaponpointer]
set max [lget $maxammo $weaponpointer]
set ammo [lget $paintammo $weaponpointer]

if { [lget $reload $weaponpointer] } {
        triggerinterval $reloader [lget $reloadinterval $weaponpointer]
        triggeritterations $reloader [/ [- 100 $ammo] 2]
        starttrigger $reloader
}

flush
}


! 5
presscript:
fireweapon


! fire
sameas: 5


! #
presscript:
if { [or [animates $imgroter] [animates $strroter] ] } { return }
next $strroter
next $imgroter
changeweapon


! paint
paintpicture src $background x 0 y 0 width $reswidth height $resheight flags "left|top"
color r 255 g 255 b 255
set height [/ $resheight 3]
paintrect x 0 y 0 width [- $reswidth 1] height [/ $resheight 3] depth 3
animation $imgroter
font size $fontsmall
paintstring value {"Frags: $frags"} x 4 y 5 flags "left|top"
paintstring value {"Deaths: $deaths"} x 4 y 25 flags "left|top"
paintstring value {"Score: $score"} x 4 y 45 flags "left|top"
animation $strroter
set height [- [/ [* $resheight 2] 3] 2]
paintrect x 50 y $height width 100 height 20 fill true color 0x990000
paintrect x 50 y $height width $health height 20 fill true color 0x336600
paintrect x 50 y [+ $height 25] width 100 height 20 fill true color 0xff0000
paintrect x 50 y [+ $height 25] width {$ammo} height 20 fill true color 0x424242
color r 255 g 153 b 51
paintrect x 49 y [- $height 1] width 101 height 21
paintrect x 49 y [+ $height 24] width 101 height 21
font size $fontsmall
color r 255 g 255 b 255
paintstring value "health:"  x 5 y $height flags top|left
paintstring value "ammo:" x 5 y [+ $height 25] flags top|left
paintstring value {"$health%"} x 100  y $height flags top|hcenter
paintstring value {"$curr"} x 100  y [+ $height 25] flags top|hcenter


! KilledMsg
set killername [hget $msgHash "player"]
set score [hget $msgHash score]
set deaths [+ $deaths 1]
cs death

! UpdateMsg
set health [hget $msgHash health]
set frags [hget $msgHash frags]
set score [hget $msgHash score]

! shownotify
set weaponpointer 0
set reload  [list false false false false true]
set ammostatus [list 13 30 4 8 0 ]
set paintammo [list 100 100 100 100 0]
set curr [lget $ammostatus $weaponpointer]
set max [lget $maxammo $weaponpointer]
set ammo 100
set health 100
resetanim $strroter
resetanim $imgroter

! hidenotify
stoptrigger $reloader true

! 1
presscript: cs death


stop}

canvas death begin{

! main
set bg [image "/icons/DeathScreen.jpg" $reswidth $resheight]
set countdown 10

! paint
paintpicture src $bg x 0 y 0 flags "left|top"
font size $fontlarge
color r 255 g 204 b 204
paintstring value {"by $killername"} x 30 y 40 flags top|left
paintstring value {"inserting in $countdown"} x [/ $reswidth 2] y [/ [* $resheight 3] 4] size
    medium color 0xffffff flags top|hcenter

! shownotify
set countdown 10
set counter [trigger interval 1000 periodic true start true itterations 10 code {
incr $countdown -1

flush
} onexit {
cs maincontrol
} ]

! hidenotify
stoptrigger $counter

stop}
```
Listing A.1: SlagMark source code

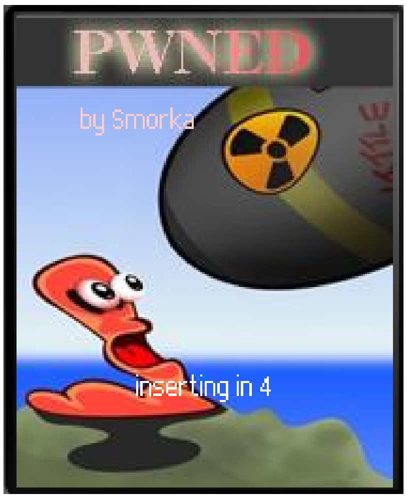
A.1.4 Screenshots

Screenshots from the SlagMark game and client:



(g) Ingame screenshot from Slagmark





(h) SlagMark game client (i) SlagMark death canvas

Figure A.1: Screenshots from SlagMark

A.2 BandHero

BandHero was developed by Morten Versvik just before the Kosmorama film festival. It is intended as a compliment to the popular game Guitar Hero. The game supports four to eight players depending on which song is played. The game builds the key pattern that the player must press based on the information in a midi file. As the timeline goes, the game will show rectangles that indicates which buttons the player should press.

A.2.1 Gameplay

The gameplay is based on pushing the right buttons at the right time. The game screen is divided into four or eight frames that display the button combination that the player must press in order to get score. As the time goes the indicators will move to the left. If the player presses the right keys when the indicators pass the leftmost edge, he will gain scores.

A.2.2 Goal of the game

Goal of the game is to achieve the highest score when the song is finished playing.

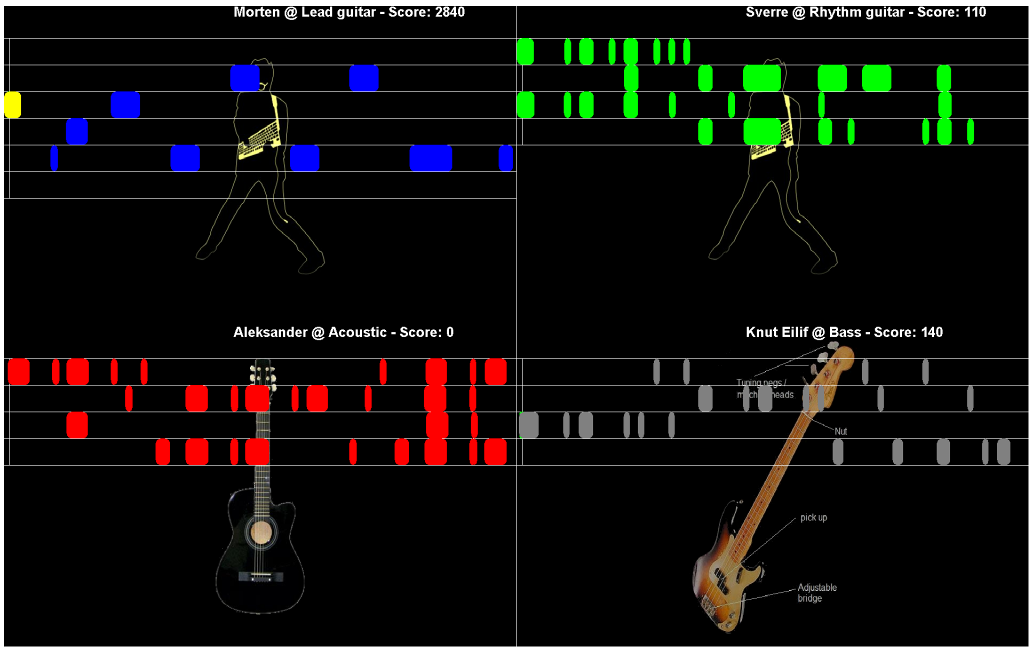
A.2.3 Client source code

```
canvas maincontrol begin{

! main
sendpressrelease 1 2 3 4
set background [image "/icons/background.jpg" $reswidth $resheight]

! setbackground
set background [image [hget $msgHash imgName] $reswidth $resheight]
flush

! paint
paintpicture src $background x 0 y 0 flags left|top

stop}
```

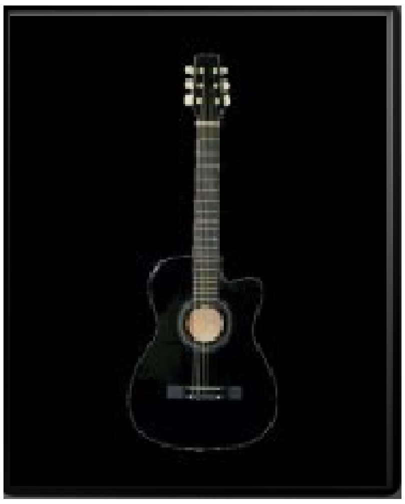
Listing A.2: SlagMark source code

A.2.4 Screenshots

Screenshots from the BandHero game and client:


Morten @ Lead guitar - Score: 2840
Sverre @ Rhythm guitar - Score: 110
Aleksander @ Acoustic - Score: 0
Knut Eilif @ Bass - Score: 140
Tuning pegs / machine heads
Nut
pick up
Adjustable bridge

(a) Ingame screenshot from BandHero



(b) BandHero game client

(c) BandHero game client

Figure A.2: Screenshots from BandHero

A.3 SelfFish

SelfFish is a top-of-the-hill game that focus on the eat or be eaten nature. The game takes place in an aquarium where every player controls one fish. The fishes gain progress by eating planktons that are dropped into the aquarium. When a player reaches a certain threshold of planktons he will grow bigger, and may eat smaller fishes.

A.3.1 Gameplay

The player is in control of one fish that can move in all directions. When a player moves to a plankton, the fish eats is and the progress bar on the client increases. When the progress bar is full, the player will increase his level and grow bigger. The player may eat fishes that are lower level. The player also has a boost function at his disposal that he might use to flee from bigger fishes or to reach plankton before others. When the player is eaten he is out of the game until the game restarts.

A.3.2 Goal of the game

The goal of the game is to be the last fish alive, to be top of the hill.

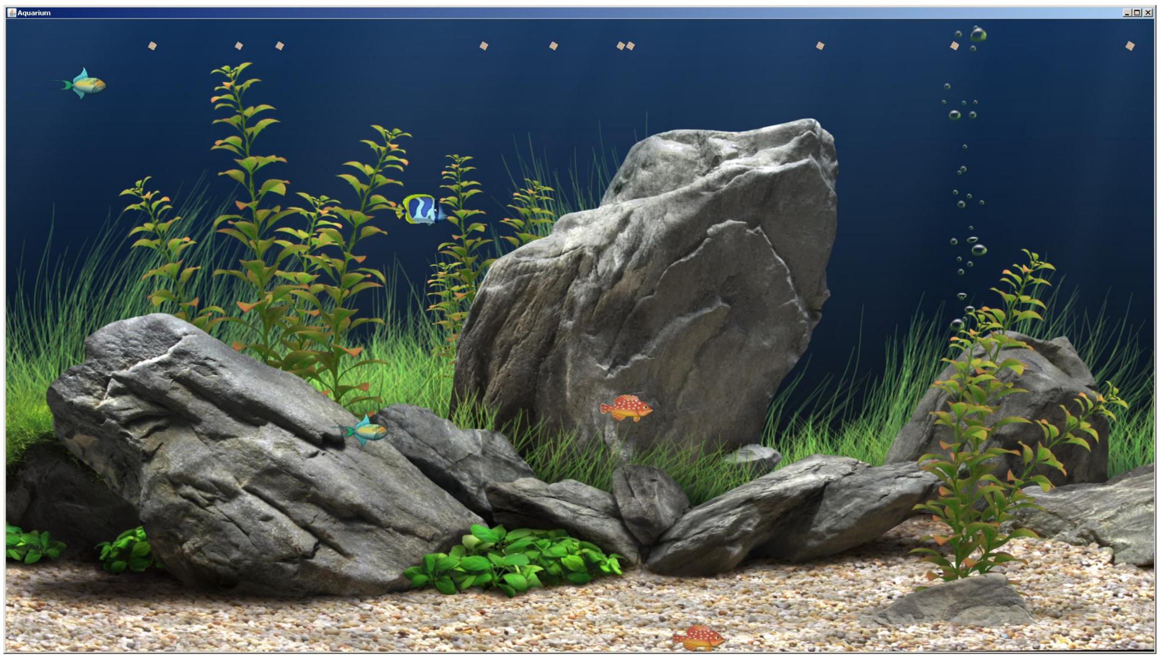
A.3.3 Client source code

```
canvas maincontrol begin{

! main
set bg [image "/fiskespill/mainBack.jpg" $reswidth $resheight]
set prog 0
set frags 0
set boost true
set level 2
set cap "ready"
sendpressrelease 1 2 3 4 6 7 8 9
set progress [progressbar x 50 y [/ $resheight 2] width 100 height 25 startpos 0 caption {"
    $prog%"}]
progcolor $progress forecolor 0x0000ff backcolor 0xff0000 middlecolor 0x00ff00 framecolor
    0x000000

set booster [progressbar x 50 y [+ [/ $resheight 2] 35] width 100 height 25 onfull {
set boost true
set cap "ready"
} onempty {
set boost false
set cap "loading"
} caption $cap]
progcolor $booster forecolor 0x0000ff backcolor 0xff0000 middlecolor 0x00ff00 framecolor
    0x000000

! paint
paintpicture src $bg x 0 y 0 flags left|top
paintstring value {"Frags: $frags } x [/ $reswidth 2] y [/ $resheight 3] flags top|hcenter
    color 0xff9963 size large style bold
paintstring value {"Level: $level } x [/ $reswidth 2] y [/ $resheight 4] flags top|hcenter
    color 0xff9963 size large style bold
paintstring value "energy" x 5 y [+ [/ $resheight 2] 3] color 0xffffff flags top|left
paintstring value "booster" x 5 y [+ [/ $resheight 2] 38] color 0xffffff flags top|left
animation $progress
animation $booster


! powerup
set frags [hget $msgHash score]
set nano [hget $msgHash energy]
set interv [- $nano $prog]
set prog $nano
incrp $progress $interv


! death
cs death

! levelup
set prog 0
incr $level
decrpi $progress 100

! 5
presscript:
if { $boost } {
decrpi $booster 100
incrp $booster 100
send boost
}
stop}

canvas death begin{

! main
set bck [image "/fiskespill/deathBack.jpg" $reswidth $resheight]
set clipx [/ $reswidth 2]
set clipy [+ [/ $resheight 3] 5 ]
set clipwidth 0
set clipheight 0

! shownotify
trigger interval 100 periodic true start true itterations [+ [/ $reswidth 2] 4] code {
incr $clipx -1
incr $clipy -1
incr $clipwidth 2
incr $clipheight 2
flush
}


! paint
paintpicture src $bck x 0 y 0 flags left|top
setclip x $clipx y $clipy width $clipwidth height $clipheight
paintstring value "OhohoPWNED" x [/ $reswidth 2] y [/ $resheight 3] flags top|hcenter color
    0xff0000 size large style bold
paintstring value "you acheived" x [/ $reswidth 2] y [/ $resheight 2] flags top|hcenter color
    0xffffff size medium
paintstring value "$frags frags" x [/ $reswidth 2] y [+ [/ $resheight 2] 14] flags top|hcenter
     color 0xffffff size medium
setclip


stop}
```

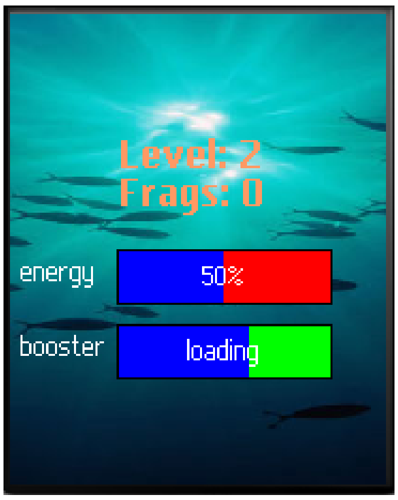
Listing A.3: SlagMark source code

A.3.4 Screenshots

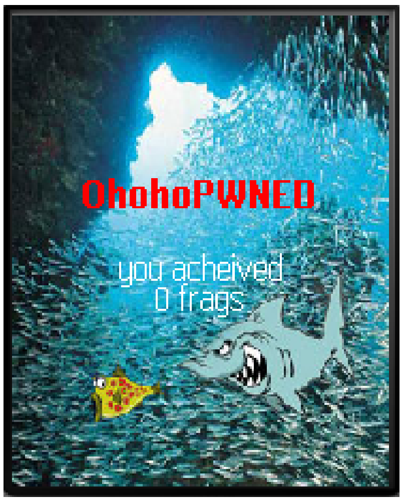
Screenshots from the SelfFish game and client:



(a) Ingame screenshot from SelfFish



(b) SelfFish game client



(c) SelfFish Death canvas

Figure A.3: Screenshots from SelfFish

APPENDIX B

Scripted client commands documentation

In this chapter we will give the grammars of the scripting platform prototype that we have developed through this project. Some of the commands will be discussed in more detail.

B.1 Grammars for the MOOSES client scripting platform

In this section we present grammars for the MOOSES client scripting platform .

```
<LoopCode> ::= <Code> | 'break' | 'continue'
<Code> ::= <Statement> {<Statement>}
<Statement> ::=  'if' '{' <Comp> '}' 'then' '{' <Statement>  '}' [ 'else' '{' <Statement>  '}'
    ]
| 'for' '{' <Statement> '}' '{' <Comp> '}' '{' <Statement> '}'
| 'foreach' <Id> <List> '{' <LoopCode> '}'
| 'while' '{' <Comp> '}' '{' <LoopCode> '}'
| 'thread' '{' <Code> '}'
| 'unset' <Id>
| 'set' <Id> [<Value>]
| 'puts' <String>
| 'rename' <Id> <Id>
| 'proc' <Id> '{' <Code> '}'
| <Id>
|'sleep' <Integer>
| 'catch' <Statement> <Id>
| return
| incr '$'<Id> [<Integer>]

<Value> ::= <Expression>|<Object>
<Object> ::= <Image>|<Stringrotator>|<Imagerotator>|<Progressbar>|<Trigger>|<Sound>
<Image> ::= 'image' <String> <Number> <Number>
<Comp> ::= <Expression> |<Cop> <Expression> <Expression>
<Expression> ::= <Term> | <operator> <Term> <Term> |
 'copy' '$'<Id> | 'eval' <statement> | 'return' [<Expression>]
<Term> ::= '$'<id>|<integer>|<Expression>
```

```
<Cop> ::= ’==’|’<=’|’>=’|’!=’|’<’|’>’|’and’|’or’
<Op> ::= ’+’|’-’|’*’|’/’|’%’
<Float>  ::= <Number>’.’<Number> | ’long’ <String>
<Number> ::= <Integer> {<Integer>} | ’rand’ | ’int’ <String> | ’abs’ <Number>
<String> ::= ’"’<char>{<char> | ’$’<Id>}’"’
<Boolean> ::= true|false|0|1|’not’ <Comp>
<Integer> ::= (Integer)
<Char> ::= (character)
<Id> ::= (atom)
```

Listing B.1: Grammars for the MOOSES Client Scripting Platform

B.2 Canvas declaration

The game application comprises a collection of canvases. The canvases are instantiated using the *canvas* command, followed by an id for the given canvas, a begin tag and a stop tag at the end of the canvas.
canvas <Id> begin{ ...stop}

To provide code to the canvas, this is done using tags between the begin and stop tag.
The tags used are:

! main - The main tag represents the init for the canvas, declaring all the variables used by the script so that the objects are stored at the Java side of the application. Objects may also be declared runtime, but it is recommended to make use of the main tag allowing the processing to take place during init.

! paint - The paint tag comprise the code that is responsible for the visual representation of the canvas. The code comprises declaration of paintcomponents and control of the controller window's Graphics object.

! <Id> - The tags that consist of an Id are used for response to server messages. The Id will be the same as the Id on the message, and the data load carried by the message is accessible through the Hecl hashtable $msghash.

! <Key> - Using a key as a tag provides for the input options to be stored on initiation of the scripts. Details on this tag may be found in Section B.7.

! shownotify - This tag holds a script that is executed when the canvas is about to get focus.

! hidenotify - This tag holds a script that is executed when the canvas loose focus.

B.3 Commands for Hecl Lists

Lists in Hecl is a collection of Strings inside a pair of '[' ']' brackets.
Declaration: <List> ::= list <String>

Operations:

llen <List> - Returns the number of elements in the list

lappend <List> {<String>} - Appends elements to the list

lset <List> <Number> [<Value>] - Sets the value of the list at the specified index, if the value is not present, the field in the list will be deleted.

lrange <List> <Number> <Number> - fetches a range of elements from list, between the two indexes

filter <List> <Id> <Comp> - The filter command takes a list and filters it according to the code provided. The current element of the list being considered is stored in the varname provided. A list of 'matches' is returned.

search <List> <Id> <Comp> - Searches for the first instance in the list that satisfy the code and return its index.

join <List> <String> - Joins the elements of a list to a string, separated by the string argument

split <String> <String> - Splits the firs string based on the delimiters provided by the second string, and puts the results into a list

lget <List> <Number> - Returns the element at the specified index

B.4 Commands for Hecl Hash tables

Hash tables support interface to the hashtables provided by Java.
Declaration: <Hashtable> ::= hash {́{́<String>} }́´
Operations:

hget <Hashtable> <String> - Returns the Object associated with the string

hset <Hashtable> <String> <Value> - Registers a new value in the hashtable

hkeys <Hashtable> - Returns the hash keys from the hashtable in a List

hclear <Hashtable> - Clears the hashtable

hremove <Hashtable> <String> - Removes the value associated with the string

B.5 Constants in the scripting platform

The scripting library provides some constants to ease the development. We refer to these numbers as constants although they might be changed from the script because they are not meant to be changed. These constants are:

$pi - Mathematical Pi

$e - Mathematical e

$resheight - Height of the screen, used for dynamic resolution adoption

$reswidth - Width of the screen

$msghash - This is not a constant, but it is not intended to be changed by the script. Variables received from the server will be put in this hashtable, which is available from the script.

B.6 Documentation on Scripted MOOSES client GUI commands

This section describes the grammars and syntax for the GUI elements supported on the scripted MOOSES client. Some of these commands are used to control the Graphics object in Java that is responsible for drawing on the canvas. These commands are:

color r <Number> g <Number> b <Number> - Which set the color of the Graphics Object. The paintcomponents that does not specify which color they will use will be painted by this color.

setclip x <Number> y <Number> width <Number> height <Number> - Sets the area which the Graphics object is allowed to draw on. Calling *setclip* without arguments will reset the clip to the screen resolution.

font size <String> style <String> - Sets the font used by the Graphics Object

flush - The flush command refreshes the screen, calling the canvas rePaint() method which triggers the Graphics Object

In addition to control of the Graphics Object, there is one other control command that concern the GUI.

cs <Id> - Changes the canvas in focus to the one indicated by the id. This will also change the additional data concerning the canvas in the MOOSEControlCanvas. Executes the hidenotify tag on the current canvas, and the shownotify tag on the canvas indicated by the id.

The rest of the commands are used to register paintcomponents to the application draw objects.

B.6.1 paintline

The *paintline* command registers a line to the canvas's paintcomponents
paintline x1 <Number> y1 <Number> x2 <Number> y2 <Number> draw <Boolean> color <String>

Field	Explanation	Default value
x1	The Lines horizontal start point	0
y1	The Lines vertical start point	0
x2	The Lines horizontal end point	0
y2	The Lines vertical end point	0
draw	Flag indicating whether the line should be drawn	true
color	Color of the line	graphics color

Table B.1: Explanations for the paintline command's arguments

B.6.2 paintstring

The *paintstring* command registers a string to the canvas's paintcomponents
paintline value <String> x <Number> y <Number> style <String> size <String> draw <Boolean> color <String>

Field	Explanation	Default value
x	x position of the string	0
y	y position of the string	0
val	the string to draw	""
size	size of the font	"small"
style	style of the font	"plain"
flags	anchoring options for the string	"left|top"
draw	flag indicating whether the string should be drawn	true
color	The color of the string to draw	"0xffffff"

Table B.2: Explanations for the paintstring command's arguments

B.6.3 paintrect

The *paintrect* command registers a rectangle to the canvas's paintcomponents
paintrect x <Number> y <Number> width <Number> height <Number> fill <Boolean> depth <Number> draw <Boolean> color <String>

Field	Explanation	Default value
x	x position of the rectangle	0
y	y position of the rectangle	0
width	The width of the rectangle	50
height	The height of the rectangle	50
fill	A flag indicating whether the rectangle should be filled	false
depth	Indicates the thickness of the rectangle in pixels	1
draw	flag indicating whether the string should be drawn	true
color	The color of the rectangle	-1

Table B.3: Explanations for the paintrectangle command's arguments

B.6.4 paintimage

The *paintimage* command registers an image to the canvas's paintcomponents
paintimage src <Image> x <Number> y <Number> draw <Boolean> flags <String>

Field	Explanation	Default value
scr	The Image which should be paint	""
x	x position of the image	0
y	y position of the image	0
draw	flag indicating whether the string should be drawn	true
flags	Anchoring options for the graphics component	"left"

Table B.4: Explanations for the paintimage command's arguments

B.6.5 animation

The *animation* command registers an animation to the Graphics object. animation <Animation> draw <Boolean>

B.7 Commands for Input

As mentioned in Chapter 19 there are two alternatives to provide input commands in our scripting platform. The Input commands may be defined in their own tag to increase the readability of the script. In addition to the tags we have provided tags to manipulate input to be able to change the input options during runtime. The keys available from the scripting platform are: <Key> ::= 0|1|2|3|4|5|6|7|8|9|fire|left|right|up|down|softl|softr|*|#
Defining input functionality using tags is done as illustrated bellow.

```
! fire # firekey
```

```
sendpress: true|false #indicates whether the key should send a KeyMessage when pressed
sendrelease: true|false # indicates whether the key should send a KeyMessage when released
sendpressrelease: true|false # indicates whether the key should send a KeyMessage on both
    press and release
presscript: <Code> # The script that is executed when the key is pressed
releasescript: <Code> # The script that is executed when the key is released
pressreleasescript: <Code> # Script that is executed both on press and release of the key
repeatkey: true|false # indicates whether the key should be repeated when pressed
interval: 1000 # changes the interval for the keyrepeater, measured in milliseconds
sameas: <Key> # makes the key reffer to the same as another key. Manipulation of the other key
    's attributes will also affect this key.
```

Listing B.2: Input Commands Using Tags

Input options may also be defined or manipulated using commands. The Input commands are explained bellow.

keyrep <Key> <Boolean> - Flag that indicate whether or not the key should repeat when pressed

sendrelease {<Key>}+ - Configure a key or collection of keys to send message when released

sendpress {<Key>}+ - Configure a key or collection of keys to send message when pressed

sendpressrelease {<Key>}+ - Configure a key or collection of keys to send message when released or pressed

presscript <Key> <Code> - Configure the script that are executed when the key is pressed

releasescript <Key> <Code> - Configure the script that are executed when the key is released

bothscript <Key> <Code> - Configure the script that are executed when the key is pressed or released

registerkey <Key> <Boolean> <Number> - Configure keyrepeater options for the given key. The first boolean indicates whether or not the key should repeat, and the number indicates the frequency of repeats.

B.8 Commands for Sound control

Sounds have commands for declaration, and playing and stopping the sound.
Declaration: <Sound> ::= sound <String>
The string is the path to the sound file.
Operations:

playsound <Sound> [<Number>] - Starts the player responsible for the given sound. The optional number in the argument indicates how many times the sound should loop, default is 0.

stopsound '$'<Id> - Stops the player responsible for the given sound.

B.9 Commands for communication

Handling communication with the script depends on which way the communication should go. For communication from the server to the client the script has to define the script in its own tag. In this tag the script may interfere with the hashtable sent from the server. It is crucial that the id of the tag and the hashes in the hashtable is equal on the game side and the script side.

```
! statusMsg
set score [hget $msghash score]
set progress [hget $msghash progress]
flush
```

Listing B.3: Communication Commands Example

The *$msghash* variable is set by the underlying MOOSEControlCanvas when receiving messages from the server.

The scripting platform also contains one command for sending data. send <Id> { <Id> ["%<Char>"] <String> }

B.10 Commands for Animation

The current scripting platform for contains three animations at this point, that are all interfaced through the same Cmds class. Some of the commands are common for all the animations.

Declaration:<Animation> ::= <Stringrotator> | <Imagerotator> | <Progressbar>

Operations:
The commands that are common for all animation objects are:

resetanim 'animation' - Resets animation objects to their initial states

speed 'animation' <Number> - Sets the speed of the animation

animates 'animation' - Returns a boolean indicating whether or not the animation is animating

The rest is specific for the different components.

B.10.1 Commands for String rotator

This section presents the specific commands for the stringrotator. **Declaration:**
<Stringrotator> ::= stringrotator x <Number> y <Number> direction <String> anchor <String>

Field	Explanation	Default value
x	x position of the stringrotator	0
y	y position of the stringrotator	0
direction	Which way the rotator should rotate, vertical or horizontal	vertical
anchor	Anchor options for the Graphics object	left|top

Table B.5: Argument explanations for String Rotator declaration

Operations:

addstring <Stringrotator> <String> - Adds a string to the stringrotator

next <Stringrotator> - Makes the stringrotator scroll to the next string

prev <Stringrotator> - Makes the stringrotator scroll to the previous string

strrotfont style <String> size <String> - Sets the font used to draw the strings

strcolor r <Number> g <Number> b <Number> color <String> - Sets the color of the strings

B.10.2 Commands for the Image rotator

This section gives the commands specific for the Imagerotator.
Declaration:
<Imagerotator> ::= imagerotator x <Number> y <Number> direction <String>

Field	Explanation	Default value
x	x position of the imagerotator	0
y	y position of the imagerotator	0
direction	Which way the rotator should rotate, vertical or horizontal	vertical

Table B.6: Argument explanations for Image Rotator declaration

Operations:

next <Imagerotator> - Makes the imagerotator rotate to the next image

prev <Imagerotator> - Makes the imagerotator rotate to the previous image

addpic <Imagerotator> <Image> - Adds an image to the imagerotator

B.10.3 Commands for the Progressbar

This section presents the commands specific for Progressbar.
Declaration:
<Progressbar> ::= progressbar x <Number> y <Number> width <Number> height <Number> startpos <Number> onempty <Code> onFull <Code> caption <String>

Field	Explanation	Default value
x	X position of the progressbar	0
y	Y position of the progressbar	0
width	Width of the progressbar	0
height	Height of the progressbar	0
startpos	Indicates the starting point of the progressbar, measured in percent	100
onempty	Code that is executed when the bar reaches 0	""
onfull	Code that is executed when the bar reaches 100	""
caption	Text which is painted on top of the bar	""

Table B.7: Argument explanations for Progressbar declaration

Operations:

incrp <Progressbar> <Number> - Increases the given progressbar the given amount

decrp <Progressbar> <Number> - Decreases the given progressbar the given amount

incrpi <Progressbar> <Number> - Increases the given progressbar the given amount without animating the bar

decrpi <Progressbar> <Number> - Decreases the given progressbar the given amount without animating the bar

progcolor <Progressbar> forecolor <Sting> backcolor <String> middlecolor <String> framecolor <Stri
- Sets the colours on the different parts of the progressbar

B.11 Commands for Triggers

Triggers are responsible for executing logic in the scripting platform for game client development, either once or continuous.

Declaration
<Trigger> ::= trigger code <Code> interval <Number> periodic <Boolean> itterations <Number> start <Boolean> onexit <Code>

Operations

Field	Explanation	Default value
code	Code to be executed by the trigger	""
interval	Sleeping interval in milliseconds when using itterations	1000
periodic	Flag indicating whether the trigger is iterative	false
itterations	Number of itterations the trigger should use, -1 means forever	0
start	Flag indicating whether the trigger should start after declaration	false
onexit	Code to be executed when the trigger if finished running	""

Table B.8: Argument explanations for Trigger declaration

pausetrigger <Trigger> - Pauses the trigger from execution

stoptrigger <Trigger> [<Boolean>] - stops the trigger. The optional flag indicates whether or not the onexit code should be executed, default is true

starttrigger <Trigger> - Starts the trigger

resumetrigger <Trigger> - Resumes a paused trigger

triggercode <Trigger> <Code> - Replaces the Triggers code

triggeronexit <Trigger> <Code> - Replaces the Triggers onexit code

triggerinterval <Trigger> <Number> - Sets the triggers sleep interval for continuous itterations

triggeritterations <Trigger> <Number> - Sets the number of itterations the trigger should execute

APPENDIX C

Abbrevations

This Chapter contains the abbrevations used in this master thesis report.

EBNF - Extended Backus-Naur Form

GUI - Graphical User Interface

CPU - Central Processing Unit

MIDP - Mobile Information Device Profile

J2SE - Java 2 Standard Edition

J2EE - Java 2 Enteprise Edition

J2ME - Java 2 Micro Edition

JAR - Java Archive

JAD - Java Application Description

PDA - Personal Data Assistant

RMS - Record Management System

API - Application Programming Interface

RAM - Random Access Memory

KVM - Kilobyte Virtual Machine

CLDC - Connection Limited Device Configuration

CDC - Connected Device Configuration

MOOSES - Multiplayer On One Screen Entertainment System

APPENDIX D

Source Code For The AFProprtyMsg

This chapter contains the sourcecode of the AFProprtyMsg used by the scripted client to transfer Data between the server and the client.

```java
package no.tellu.common.javaframe.messages;

import java.io.*;
import java.util.Enumeration;
import java.util.Hashtable;

public class AFPropertyMsg extends ActorMsg {

    public String msgId;
    private Hashtable property;
    private Object o;


    public AFPropertyMsg() {
    }


    public String getMsgId() {
        return msgId;
    }

    public AFPropertyMsg(ActorMsg am, String msgId) {
        this.setSenderRole(am.getSenderRole());
        this.setReceiverRole(am.getReceiverRole());
        this.msgId = msgId;
    }


    public AFPropertyMsg(String msgId) {
        this.msgId = msgId;
    }

    public boolean isInstanceOf(String name) {
        return (msgId != null && msgId.equals(name));
    }

    public Hashtable getProperty() {
        return property;
    }

    public Object getProperty(String name) {
        if (property == null) return null;
        o = property.get(name);
        return o;
    }

    public String getString(String name) {
        if (property == null) return null;
        o = property.get(name);
        if (o instanceof String)
            return (String) o;
        return null;
    }

    public int getInt(String name) {
        if (property == null) return 0;
        o = property.get(name);
        if (o instanceof Integer)
            return ((Integer) o).intValue();
        return 0;
    }

    public boolean getBoolean(String name) {
        if (property == null) return false;
        o = property.get(name);
        if (o instanceof Boolean)
            return ((Boolean) o).booleanValue();
        return false;
    }

    public void setProperty(String name, Object value) {
        if (value == null)
            return;
        if (property == null) {
            property = new Hashtable();
        }
        property.put(name, value);
    }

    public void setBoolean(String name, boolean value) {
        if (property == null) {
            property = new Hashtable();
        }
        property.put(name, new Boolean(value));
    }


    public void setInt(String name, int value) {
        if (property == null) {
            property = new Hashtable();
        }
        property.put(name, new Integer(value));
    }

    public String getSignalName() {
        return msgId;
    }


    public String messageContent() {
        StringBuffer sb = new StringBuffer();
        sb.append("MsgId: " + msgId);
        if (property != null) {
            Enumeration en = property.keys();
            while (en.hasMoreElements()) {
                String s = (String) en.nextElement();
                sb.append(s + ": " + property.get(s) + " : ");
            }
        }
        return super.messageContent() + sb;    //To change body of overridden methods use File
             | Settings | File Templates.
    }

    public AFPropertyMsg(InputStream is) throws IOException {
        DataInputStream din = new DataInputStream(is);
         msgId = StringHelper.resurrect(din);
        property = HashtableHelper.resurrect(din);
    }

    /**
     * This function implements the serialization of the object.
     *
     * @return a byte array with the objects data
     * @throws java.io.IOException
     */
    public byte[] serialize() throws IOException {
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        DataOutputStream dout = new DataOutputStream(bout);
        dout.write(super.serialize());
        dout.write(StringHelper.persist(msgId));
        dout.write(HashtableHelper.persist(property));
        dout.flush();
        return bout.toByteArray();
    }

    /**
     * Use this function for resurrection of the object
     *
     * @param data The serialized data containing the object data
     * @throws java.io.IOException
     */
    public ByteArrayInputStream deSerialize(byte[] data) throws IOException {
        ByteArrayInputStream bin = super.deSerialize(data);
        DataInputStream din = new DataInputStream(bin);
        msgId = StringHelper.resurrect(din);
        property = HashtableHelper.resurrect(din);
        return bin;
    }
}
```

Listing D.1: AFPropertyMsg class

Part VIII

Bibliography

Bibliography

[1] Kosmorama - trondheims internasjonale filmfestival. "http://www.kosmorama.no", 2007.

[2] Sun Microsystems. The java me platform, 2007.

[3] Sumi Helal. Pervasive java. Technical report, University of Florida, 2002.

[4] Alf Inge Wang. Forskningsprogrammet dataspill. http://www.ime.ntnu.no/forskning/prosjekter/dataspill, 26.04.2007.

[5] Morten Versvik and Alexander Baumann Spro. Multiplayer on one screen - co operative gaming. Master's thesis, Nowegian University of Science and Recnology, Spring 2007.

[6] Audun Kvasbø. Mooses game concepts. Master's thesis, Norwegian University of Science and Technology, 2007.

[7] Morten Versvik Sverre Morka and Alexander Baumann Spro. Multiplayer on one screen. Master's thesis, Nowegian University of Science and Technology, 2006.

[8] Tellu and NTNU. Mooses homepage. "http://www.mooses.no", 2006.

[9] Tellu. Tellu homepage. "http://www.tellu.no", 2006.

[10] Victor R. Selby Richard W. Rombach, H. Dieter Basili. Experimental software engineering issues: critical assessment and future directions, pages 3–12. Springer, first edition, 1993.

[11] The Eclipse Foundation. What is eclipse. "http://www.eclipse.org/org/", 2005.

[12] Open Source Technology Group. Texlipse. "http://texlipse.sourceforge.net/", 2005.

[13] Open Source Technology Group. Eclipseme. "http://sourceforge.net/projects/eclipseme/", 2005.

[14] Christian Schenk. About miktex. "http://www.miktex.org/about.html", 2005.

[15] Sony Ericsson Mobile Communication AB. Java docs tools. "http://developer.sonyericsson.com/site/global/docstools/java/p_java.jsp", 2006.

[16] IBM. J2me record management store. "http://www-128.ibm.com/developerworks/library/wi-rms/", 2002-01-05.

[17] Sony Ericsson. Sonyericsson k800. "http://developer.sonyericsson.com/site/global/products/phonegallery/k800/p_k800.jsp", 2006.

[18] Chamseddine Talhi Mourad Debbabi, Mohamed Saleh and Sami Zhioua. Embedded Java Security. Springer London, 2007.

[19] Bill Venners. Bytecode basics. "http://www.javaworld.com/javaworld/jw-09-1996/jw-09-bytecodes.html", 1996.

[20] Rolv Bræk, Geir Melby, and Knut Eilif Husa. Serviceframe whitepaper. Technical report, Ericsson, 2002.

[21] Geir Melby and Knut Eilif Husa. Actorframe developers guide. Technical report, Ericsson, 2005.

[22] Oxford Reference Online Premium. scripting language. "http://www.oxfordreference.com/views/ENTRY.html?subview=Main&entry=t11.e4636&authstatuscode=202", 2004.

[23] John K. Ousterhout. Scripting: Higher level programming for the 21st century. Technical report, Tcl Developer Xchange, 1998.

[24] Wikipedia. Scripting language. http://en.wikipedia.org/wiki/Scripting_language, 2006-20-09.

[25] Batch files for dos, os/2, windows 95/98, nt 4, 2000 and xp. "http://www.robvanderwoude.com/batchfiles.html".

[26] Adobe. Actionscript technology center. http://www.adobe.com/devnet/actionscript/.

[27] The official u.s. army game. "http://www.americasarmy.com".

[28] Americas army game manual. "http://manual.americasarmy.com/index.php/Main_Page".

[29] Official php website. "http://php.net".

[30] Smx: Server macro expansion. "http://www.smxlang.org/index.html".

[31] Mary Bellis. The history of javascript. "`http://inventors.about.com/od/jstartinventions/a/JavaScript.htm`", 2007.

[32] Alan Finn. Vbscript fundamentals for windows scripting - the basics. "`http://www.2000trainers.com/windows-scripting/vbscript-windows-scripting-1/`", April 2003.

[33] Elaine Ashton. The timeline of perl and its culture. `http://history.perl.org/PerlTimeline.pdf`, 2002.

[34] Brian W. Kernighan Alfred V. Aho and Peter J. Weinberger. The AWK Programming Language. Addison-Wesley, 1988.

[35] Peter Van Roy and Seif Haridi. Concepts, Techniques, and Models of Computer Programming. The MIT Press, 2004.

[36] George H. Forman and John Zahorjan. The challenges of mobile computing. Technical report, University of Washington, 1994.

[37] Simon Whiteside. Threenode format. "`http://www.simkin.co.uk/Docs/Simkin/TreeNode.html`", 2001.

[38] Simon Whiteside. Simkin - the embedable scripting language. "`http://www.simkin.co.uk/Features.shtml`".

[39] Ty Coon. Gnu lesser general public license. "`http://www.gnu.org/licenses/lgpl.html`", February 1999.

[40] Adobe Systems Incorporated. Postscript language reference third edition, February 1999.

[41] Phonescript homepage. "`http://phonescript.org`".

[42] Edgewall Software. Trac license. "`http://phonescript.org`", 2003.

[43] David N. Welton and Wolfgang Kechel. Hecl. "`http://www.hecl.org/`".

[44] he Apache Software Foundation. Apache license, version 2.0. "`http://www.apache.org/licenses/LICENSE-2.0`", January 2004.

[45] Sun Microsystems Inc. Java native interface. "`http://java.sun.com/j2se/1.4.2/docs/guide/jni/`", 2003.

[46] Mark J. P. Wolf. The Medium of the Video Game,. University of Texas Press, 2001.

[47] Shooter game. "`http://en.wikipedia.org/wiki/Shooter_game`", 2007.

[48] Christopher Wiliams Mark Burge. Midp 2.0 changing the face of j2me gaming. Technical report, Armstrong Atlantic State University, 2004.

[49] Wikipedia. The lost vikings. "`http://en.wikipedia.org/wiki/The_Lost_Vikings`".

[50] Blizzard Entertainment. "`http://www.blizzard.com/blizzclassic/#lostvikings`", 2006.

[51] PlanetHalfLife. Counter strike. "`http://planethalflife.gamespy.com/cs/`".

[52] Electronic Arts. Battlefield 2 product description. "`http://www.ea.com/official/battlefield/battlefield2/us/features.jsp`", 2005.

[53] Carl-Henrik Wolf Lund Alf Inge Wang, Michael Sars Norum. Issues relatd to development of wireless peer-to-peer games in j2me. Technical report, Norwegian University of Science and Tecnology, Bearingpoint, Bekk Consulting AS, 2005.

[54] Paul Clements & Rick Kazman Len Bass. Software Architecture in Practise. Addison-Wesley Professional, 2 edition, 3 2003.

[55] Ruth Malan and Dana Bredemeyer. Architecture resources for enterprise advantage. Technical report, Bredemeyer Consulting, 2001.