Swords and Sorcery Implementation Description
University of Idaho, CS 383, Spring 2014
May 9, 2014

1 Introduction and Document Description

This is a description of the Swords and Sorcery implementation, both in general and the implementation of major code modules.

This document particularly describes what was not covered in the design documentation, as well as places where implementation differs from design.

Sections are:

- Implemetation Overview
- Deployment, System Requirements, and Libraries
- Building
- Major Code Modules

2 Implementation Overview

Swords and Sorcery is programmed in Java 8 (after switching from Java 7), and is cross platform across Windows, Mac and Linux.

It uses a 2D top-down view and graphics based on the physical game by SPI, has a GUI programmed in JavaFX and swing, and TCP/IP networking support.

Unfortunately the implementation remains incomplete, so while some basic functionality exists, users will find it extremely difficult to face each other in battle over the internet.

What works or partially works (to the best of our knowledge) includes:

1. Starting a new game (partial)
2. Selecting a Scenario
3. End Inter-Phase
4. Move
5. Teleport
6. Select Unit From Stack
7. Attack
8. Advancing Units (partial)

9. Retreating (partial
10. View Unit Statistics
11. Spellcasting (partial)
12. Starting a client
13. Starting a server
14. Connecting to a server and communicating back and forth
15. Tagged Chat: Lobby-wide and Server-wide
16. Show online users by lobby/server
17. Lobbies: show, create, join, leave
18. Viewing Army Units, Terrain, or basic Hex Edges on the Main Map
19. Viewing a blank Diplomacy Map
20. Seeing the solar display or game phases change when ending a turn

What does not work (to the best of my knowledge) includes:

1. Join Game
2. Set Up
3. Resume Game
4. Save Game
5. Select Ally Candidate
6. Carry
7. Choose Leader
8. Spend Combat Manna
9. Rally
10. Manna Transfer
11. Diplomacy or Emissaries or Prisoners
12. Sacrifice
13. Dragon Blockades
14. Game State Synchronization between clients
15. Special Hex Rendering

16. Character or Monster Rendering
17. MiniMap or Diplomacy Map
18. Random Events
19. User to user chat

3 Running, Deployment, System Requirements, and Libraries

Swords and Sorcery requires the Java 8 JRE installed to run. This must be installed by the user independantly before trying to launch the game.

There is no installer package. The program takes the form of two .jar files (one for the client, and one for the server), along with a library folder and a resources folder.

In Windows the user should be able to launch the program by double clicking one of these .jar files.

In other supported operating systems the user should be able to launch the program with Java from the command line, with one of the following commands to launch the client or server respectively:

1. java -jar client.jar
2. java -jar server.jar

The program must be launched from the command line in this manner if the user wishes to view debug output.

The only required libraries are

- json-simple 1.1.1
- controlsfx 8.0.5

.jar distributions of these two libraries are included.

Swords and Sorcery isn't the fastest Java code on the block, but should be runnable by most contemporary systems running Windows, Mac, or Linux.

The folder for a swordsorc distribution should contain the following:

1. lib - containing library jars
2. resources
3. client.jar
4. server.jar

4 Building

Swords and Sorcery is hosted on github. If git is installed then the source can be downloaded with the following command:

git clone https://github.com/cjeffery/sworsorc.git

Swords and Sorcery can be compiled from inside Netbeans 8 by selecting either the ¡default config¿ or the Game build targets to build the client. The server can be built by selecting the Server target.

Other build targets exist for debugging or legacy reasons and should be ignored.

Alternatively Swords and Sorcery can be built from the command line with ant, using a build script similar to the following:

ant -f build.xml -q -Dconfig=Game jar ant -f build.xml -q -Dconfig=Server jar

to build the client and server respectively.

Be aware that however the project is built, both client and server will build a file called sworsorc.jar

If both client and server have to exist side by side, then the .jar files will have to be renamed to avoid overwriting eachother.

5 Major Code Modules

Swords and Sorcery has several major code modules. Amongst them the following have interesting enough implementations to bear discussing:

- The JavaFX HUD
- The Map Rendering and the Map View Code
- The Networking System

- The Map File Format
- The Movement Calculator
- Various Gameplay Classes

6 The JavaFX HUD

JavaFX was chosen over swing for the GUI because it's newer and more active than swing. And because it supports scaling natively (as opposed to needing to use kludges to scale stuff in swing).

JavaFX organizes user interfaces into Stages and Scenes. A stage displays a scene. Very theatrical.

JavaFX uses .fxml files to describe interfaces. These can be thought of as similar to HTML.

mainMenu.fxml describes the layout for the Main Options menu which is the first menu that is seen on program launch.

Likewise hud.fxml describes the layout of the actual game screen.

JavaFX also has classes called Controller Classes, that determine the behavior of programs that use these layouts.

In our project MainMenuController.java controls the Main Menu, and is in charge of, for instance, calling a function that actually starts the game when the user hits the "Start Game" button.

HUDController likewise controlls the Main Game screen and has several notable features:

- Load the game map into it's panel
- Update the Sun Position Display at the end of every turn
- Select units when the map is clicked on. And display their stats
- Send messages from the chat box to the server, and vice versa

7 JavaFX MapView interaction

The GUI was switched to JavaFX partway through the project.

This proved a challenge for the Map View implementation, as there are unique challenges using swing widgets in JavaFX based code.

Passable support was fortunately very doable using the SwingNode class.

But one unresolved implementation issue was threading concerns.

Any interaction between swing and JavaFX has to be done on the appropriate thread. But there wasn't enough time to resolve these threading concerns.

The code seems to work OK, but this bug could conceivably lead to weird or inexplicable behavior on certain systems or edge cases.

8 The Map Rendering Code and the Map View

None of the Map Rendering code or Map View code was designed. It was just pulled out of thin air, using the time-honored alchemical process of spontaneous generation.

Fortunately this worked pretty well, as Java's libraries dictate a lot of the design up front, and I (Colin) have had some experience in the past with Java swing programming.

8.1 The Map Rendering Code

There are a few questionable design decisions, and a bit of duplicate code, but overall it worked pretty well and the implementation didn't end up inherently broken.

The Map Rendering code is done with Java's swing framework. All rendering is done onto standard swing Graphics2D objects that are passed in from the Map View.

It consists of two main classes that work in similar ways:

1. UnitPainter
2. HexPainter

Both of these classes are similar, and if they were ever refactored it might make sense to have them both inheret from the same class.

Both classes have methods that accept a Graphics2D object from Java's 2D API to render on to.

Both classes load a variety of PNG images from disk to assist in rendering. Most of the terrain images were drawn by Joe Higley. The unit images were drawn by Colin Clifford.

HexPainter is in charge of rendering Hexagons and edges, and passes off Unit rendering to UnitPainter.

Unit Rendering involves drawing a partially transparant unit icon, and (in the case of Army Units) a status string.

Due to time constraints only Army Units are mostly supported. With characters being stubbed in and looking weird.

Edges are rendering using simple straight colored lines using swing. There's a bit of hexagon math to prevent two elements of the same edge from obscuring eachother, and to draw roads / trails from the center of the hexagon to the edge.

8.2 The Map View Code

Most of the Map View code is in a single class called (appropriately enough) MapView.

MapView is a Swing widget, but the GUI is JavaFX based. So the GUI wraps MapView inside a SwingNode object.

MapView does two main things:

1. Implement various swing widget methods (most notably PaintComponent)
2. Contain methods for necessary math for determining hexagon tiling

MapView implements scrollable, so it can be added to a JScrollPane to support scrolling.

The PaintComponent method is what interacts with the Map Rendering code.

The hexagon tiling math is interesting

8.2.1 Hexagon Math

Hexagon math was cobbled together from various sources on the internet and frenzied scribbles on note-paper.

It's honestly pretty hairy, so I tried to include helpful comments in MapView.

I will not fully describe all the hexagon math here, but I'll share the general principles.

Hexagons are oriented such that the top and bottom are horizontal lines, as per the physical Swords and Sorcery map.

The width of one hexagon is 64px. But as columns fit together smoothly the distance between a column and it's neighbors is 3/4ths that.

The height of a hexagon is the radius (32px) times the square root of 3.

But the key to hexagon tiling was this stackoverflow post: http://gamedev.stackexchange.com/a/20762

Which is to say, Hexagon picking involves converting between hexagon shaped and rectangle shaped tilings. Pretty cool.

With this I was able to tile hexagons graphically, and convert mouse coordinates to hexagons or hexagon edges.

8.3 Interaction Between MapView and Map Rendering

The Map Rendering code has three entry main entry points:

1. Draw a hexagon
2. Draw a hexagon edge
3. Highlight a hexagon

Each of these take a Graphics2D object that is assumed to be pre-translated so that the Rendering Code only has to draw off of the origin.

The first two cases used to be merged, but it was important to avoid drawing a given edge twice, and to avoid an edge being obscured by a neighboring hexagon.

So MapView takes the following passes:

1. Requests the Map Rendering code to draw all the hexagons
2. Requests the Map Rendering code to highlight any hexagons that need highlighting
3. Requests the Map Rendering code to draw all the edges

This "layered" approach gives a pretty smooth look, where nothing draws on top of stuff it shouldn't too much.

9 The Networking System

The goal was to implement network communication of state changes, basic lobby funcationality, and basic chat.

Since we did not integrate our design with the rest of the game at the beginning, there was no logging subsystem setup to track any and all changes to game state. Therefore, our implementation utlimatly involved trying to get individual parts working by hardcoding messages in the requisite areas of code.

9.1 Networking Model

The Networking model used is the infamous client/server model established by Quake. The basic model is thus: clients who want to play together connect to a server, and the server facilitates the connection between them, and sometimes takes on the processing chores of clients as well.

In our implementation of the model, there are "thick" clients that handle almost all of the processing chores, with the server's primary job being to forward messages between clients and facilitate network functionality such as Lobbies, Chat, and Polling. This is similiar to the peer to peer model used by RTS games, with thick clients processing messages recieved, but just with a server inbetween.

Of course, this opens more opportunies for cheating than I want to think of, and an incredible amount of vulnerabilities. If this game were to hypothetically go mass-market, there would need to be a major overhaul before release, for the sake of user security(not to mention online game quality)

9.2 Messages

Network Messages consist of the following:

- FLAG: message category (ex: GAME or CHAT)
- TAG: specific type of message (ex: LOBBY combined with a CHAT flag for Lobby chat)
- Sender: String specifying who sent the message, either the handle of a user, global server, or a specific sub-system
- Data: a List of POJOs (Plain Old Java Objects) which encapsulates any messages being sent. This is a semi-rough hack to allow for (almost) any class to be sent over the network, which we needed with units, spells, and the like

This was better than sending strings, since creating the command parsing and execution framework, where strings send over the network execute commands on the client recieving to change the state, would've possibly required a complete redesign of the core code.

It was not, however, anywhere near decent. The major issues we ran into were parsing the list of data, which led to null pointer exceptions and strange output, and the fact

that any object sent needed to be Serializable. JavaFX, unfortunatly, is not, and this led to exceptions and terminated connections whenever a class with a JavaFX object was attemped to be sent over the network.

The message format did serve its purpose, however, and we were able to get some limited funcationality using it.

9.3 Threaded Connections

From the beginning, the network design included theads. Initially, it was simply a sending thread and a recieving thread. However, that expanded into the use of sending/recieving, and command processing threads client side, and sending/recieving threads dedicated to each unique client connection on the server side.

Each thread handles one "half" of the connection, so sending thread would write to the socket, and the recieving thread would read from the socket. Modern ethernet networks support two-way communication using switches, so the old 10Base2 Bus model is certainly no longer viable. Token ring was briefly considored, but would cause more syncronization issues than it would solve, again considoring the technology available.

Queues were implemented to pass messages in and out of threads, basically acting as a "buffer", so if there were slow-downs(common occurance in networks), the threads wouldn't be causing data to be dumped.

The threads themselves didn't cause an abnormal amount of issues, and served their job well. If this were to be put on, again, the "hypothetical market", there would need to be changes to addres performance. Since (far as chris understand it) the threads are a client library, they aren't fully utilizing concurrence available by using OS level threads. Not to mention queue sizes and network syncronization, and the always-looming threat of a buffer overflow.

9.4 The Network Client

This is the Meat and Potatoes of the Networking system. It is a static singleton class, and handles all incoming/outgoing messages, user command processing, creating/closing connections, and anything else related to networking.

There are many public utility functions, such as startGame, endTurn, and generateUniqueID. Most method calls, however, are to the send method, and its varients (sendChatMessage, poke). This sends a message including a flag, tag, the username of that client, and any data passed to it through var-args. This allows for quite a bit of flexibility in calls, and leaves the processing of a message up to its arguments, not the method call itself.

Most of the difficulty in the implementation of the client was with command/message processing. The command thread processes any user input to the console box, such as network commands or chat messages, and sends the relevant messages. That was just a very hairy if-else mess, but not too terrible beyond constant maintence whenever a command changed.

The ClientRecieverThread, on the other hand, was troublesome. Its structure is almost an identical copy of its twin in ClientObject server-side. This meant a LOT of maintence whenever something changed, and lots of hairpulling resulted from the errors resulting from these subtle differences. This probably could've been done with a fancy design pattern during the design phase, but it was overlooked, and that led to a lot of extra time spent implementing.

9.5 The Network Server

This is actually simplier than the client side, even though it is utilizing more classes than the client side.

The Network Server itself is a static singleton class, with a constantly-running loop that listens for and recieves client connections. It manages the lobbies as a list of Lobby objects, and connected clients as a list of ClientObjects (each with their own pair of threads).

The primary functions of the server, beyond just passing along messages, is handling the "nitty-gritty" details of the network system. These include global broadcasts, connections/disconnects, and the mangement of lobbies. It does not care about the game state in any way.

That is the lobby's job. Each lobby object manages its own pool of clients, with a lobby having at least one client at any point in time. Lobby-wide broadcasts are handled here, as well as phase changes and game turns.

Each client connection that is recieved is assigned to a client object, each with its own name (players username) and unique ID. These objects handle all of the server's incoming and outgoing communication, and anything being sent out must go through a client object. Thusly, broadcasts involve looping through the client lists and calling the send function on each one. Additionally, they have the responsibility of processing ALL incoming messages, and calling the requisite methods in NetworkServer or one of the Lobby objects.

9.6 The Conductor

The Conductor is really just that: a "network train" conductor Anything game related that is the clients responsibility is passed to the conductor's processMessage method,

which then invokes the requisite methods by using static instances of UnitPool, HUDController, and others.

In its most basic form, its just a big switch statement that either directly calls the methods, or calls class utility methods that then directly call the methods on the instances.

The only major issue with it is the same as with the recieving threads: any command change requires mantence accross the entire network codebase. Not very modular, and very very brittle.

In spite of that, it does work, and serves its purpose in this instance.

10 The Map File Format

The MainMap data is stored in a large XML file

It has the following information including

- Hex ID
- Whether it's a capital, town, city, or castle hex
- Whether it's a town hex
- Neighboring Hex IDs
- Province Name
- Whether it's a vortex or portal
- What sorts of hex edges there are
- Whether the hex has a name

11 The Movement Calculator

The MovementCalculator class has 12 methods and two static fields. The static fields are for optimization of movement and retreating. The optimization of movement uses the allowanceCache HashMap to hold only the fast routes that the method getValidMoves(...) finds. getValidMoves(...) is a public static method that recursively populates an ArrayList of MapHex objects based on constraints that apply to the unit that is moving. These constraints include terrain types, hex edges, enemy occupation and zone of control, and the movement allowance allotted to the unit per turn. The hashmap that optimizes the getValidMoves(...) method holds each valid MapHex that the unit can reach as a key

and the unit's remaining movement allowance for moving into that hex. This allows a dynamic movement system, where a player may move a unit twice or more in a movement phase, while the unit still remains in the bound of its movement allowance. The other field in MovementCalculator, retreatAllowanceCache, serves a similar purpose as allowance cache, but is used by the retreat methods. Retreating is also done recursively, and all valid moves for a retreating unit are generated and added to the hashmap, retreatAllowanceCache. From there, the hash map is filtered according to the S&S rules for retreating from combat. There is a stated hierarchy of locations that a unit can retreat to, which the method filters for and follows, ultimately generating and returning a list of hexes that a unit can retreat to. If the size of the ArrayList returned is zero, or if the ArrayList is null, then the unit has no valid retreat locations, and should be destroyed. The remaining methods that are undescribed here are simply helper methods for the main methods, or wrappers, as in the case of wrapperMovement(...).

12 Gameplay Model Classes

12.1 Army Units

When implementation came around we discovered that having sub-classes for every different army unit sub-type was not practical. Instead a single army unit is with different flags to describe the unit is used. The main super class of movable units is still used with the sub-classes army unit, character, and monster but then stops there. Also a race and nation variable unit had to be added to the class in order for things like the movement calculator to work.

12.2 Characters

Characters have a series of variables needed for spell casting, leadership, and diplomacy. Something new is the idea of a spell book, a class that reads it's character's information to determine what spells can be cast. In order for this class to work properly the character class contains a spell book and the spell book contains its corresponding character. In order for the scenario reader to work properly a character creator was made in the character class that takes the name of a character, finds the matching data on the character, and creates a character with the data.

12.3 Spells

For spells to work properly they needed to exist separate from the main program. This was implemented by creating a separate GUI for spells. When a character "casts" a spell, the spell book class is called upon which created a GUI with all spell options that character has. Once a spell is chosen the user has the option of reading the details of the

spell or casting the spell.

Once a spell is "cast", the specific spell class will be called and ask to select target(s) to cast. After a target is selected, limitations will be checked to see if succeed to cast the spell or not. If succeed, the effects of the spell are added to the games state. If not, there are two choise: "Recast" depending on different limitations or "OK" to terminate current spell implementation. Whatever the spell is cast or not, options that cast other spells or end spell phase are available.

12.4 Combat

12.4.1 LaunchCombat

The LaunchCombat functin first takes inputs Movable units, this connects with the HUD-Controller class: left mouse click to get selected stack, and right mouse click to get target stack. When a player in the combat phase, they should pick both units then press $'A'$ to launch the combat. When clicked A button, first it shows dialogs to ask the attacker player if they want to add friendly units which surround the defender units into the combat, and then it will pop out units detail include Attackers strength, Defenders strength, Defenders Terrain type, and Defenders strength after Terrain Type bonus with a confirmation dialog. The certain player should click yes button for showing combat result, or click no for doing nothing. If the player clicks yes button, the Combat result will shows as a notification on the right bottom conner, with this result both attacker player and defender player may meet three situations: Nothing Change, Retreat, Elimination. If the combat result returns 0, then the certain player does not need to do any reactions from this combat, if the result is a negative number, the certain player should eliminate numbers(since result is a integer number) of units from the units list; if the result is a positive number, then the player should decide to eliminate the unit OR retreat the unit.

12.4.2 Retreat

First the result came from the LaunchCombat, so the player can get its result value(negative, positive integer, or zero), if the result is a positive number then the player should decide to retreat the units or not, if not, then they should eliminate the unit, if yes, execute the Retreat function. The retreat function will send necessary input for the method getRetreatMove in the class Movement Calculator, so it can get which hexes that the certain units can move to. And it will highlight those available hexes with red color, then the units will be randomly assigned to move to an available hex.

12.5 Unit Pool

Unitpool wasn't in the design.

It was added as a design modification during one of our sprints. The rules of the game required tracking of units, manipulation of persistent units, a way to create, and destroy those units. The class name is UnitPool.

12.6 HexStack

HexStack was originally going to be named Stack, and a component of a Hex.

In the implementation it exists to aggregate units in the UnitPool.

It's named HexStack instead of Stack to avoid a name conflict with the Java library.