



**Swords and Sorcery Implementation Description**  
**University of Idaho, CS 383, Spring 2014**  
**May 6, 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:

- Implementation Overview
- Deployment, System Requirements, and Libraries
- Building
- Major Code Modules
  - The JavaFX HUD
  - The Map Rendering Code
  - The Networking System
  - The Map File Format
  - The Movement Calculator
  - ...

## 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 some TCP networking support.

Unfortunately the implementation remains incomplete, so while some basic functionality exists, users will find it impossible to face eachother in battle over the internet.

What does work includes:

TODO: add to this

1. Starting a client
2. Starting a server

3. Connecting to a server and communicating back and forth
4. Loading a predetermined scenario
5. Moving units according to their movement allowances
6. Casting basic spells
7. Teleporting
8. Viewing Unit statistics
9. Viewing Army Units, Terrain, or basic Hex Edges on the Main Map
10. Viewing a blank Diplomacy Map
11. Starting a server
12. Seeing the solar display or game phases change when ending a turn

What does not work includes:

TODO: remove from this

1. Game State Synchronization between clients
2. Special Hex Rendering
3. Character or Monster Rendering
4. MiniMap or Diplomacy Map
5. Full retreating or advancing
6. Random Events
7. Starting a server

### **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.

## 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

Describe Game.java and HUDController.java I guess

## 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, as implemented by TODO: WHO DID THIS?

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 inherit 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 transparent 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 rendered using simple straight colored lines using swing. There's a bit of hexagon math to prevent two elements of the same edge from obscuring each other, 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  $\frac{3}{4}$ ths 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 rect-angle 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

Network team should write this

## 10 The Map File Format

## 11 The Movement Calculator

Be sure to mention that it uses a recursive algorithm



## 12    **Gameplay Model Classes**

Gameplay team knock yourself out.

UnitPool could also go here, or in it's own section.