Blockade! Game for Android 1.5

James Marshall

December 13th, 2010

1 Overview

Blockade! is a hex-tile based game for the Android 1.5 platform. The user is tasked with navigating a small ship to different ports to earn points while avoiding enemy ships. The graphics are 2D, and sprite based.

2 Platform

The Android OS was chosen as the target platform due to it's ease of programming and capabilities. Programing can be done entirely in Java Standard Edition, and there is a large array of supporting libraries. The desirable capabilities include larger screen resolutions (compared to many J2ME devices) and a touch screen.

The decision to target the older 1.5 version greatly increases the potential user base to include those with first generation android devices. For a 2D game, I found no need for the features introduced by the later versions of the OS.

3 Design

The final design remains for the most part true to the design laid out in the project proposal, with a few additions and missing components. The removal of components was mainly a result of the scaled back feature set, and the immaturity of the features implemented. For example, currently there is no sound in the game and the AI was too simplistic to warrant it's own class. The additions where made to accommodate the Android model and in an attempt to winnow out the core Map class as a reusable component, which I will discuss later.

3.1 Class Structure

Please refer to Figure 1 for the class structure. Block arrows indicate inheritance, thin arrows indicate communication.

The section labeled "Intents" encompasses the classes that together make up the menu system for the game. Each has an associated layout which is displayed to the user (for "Play" this ends up being the main game). Each is launched by an Intent, which is a message passing structure created for Android. These Intents can be external, such as the one from the OS which launches the main menu, or internal, such as the intent started when the player dies to switch to the top scores layout.

The Play class is our main entry point into the game, which is represented by the Blockade class. Blockade controls all of the logic of the game and also launches a thread to control the drawing and updating of the game. Blockade also reads in the data for the level from a file, which it then uses to instantiate the Map class.

The Map class is responsible for drawing and updating the game map, and also keeps track of all the Tiles and Actors in the game (arrows to indicate communication were omitted). It also provides useful functions such as a distance calculator and a mechanism for setting a tile as selected. For right now, the library distinction is only made by convention: the source code for the Map, Tile, and Actor class is in the same source tree as the rest.

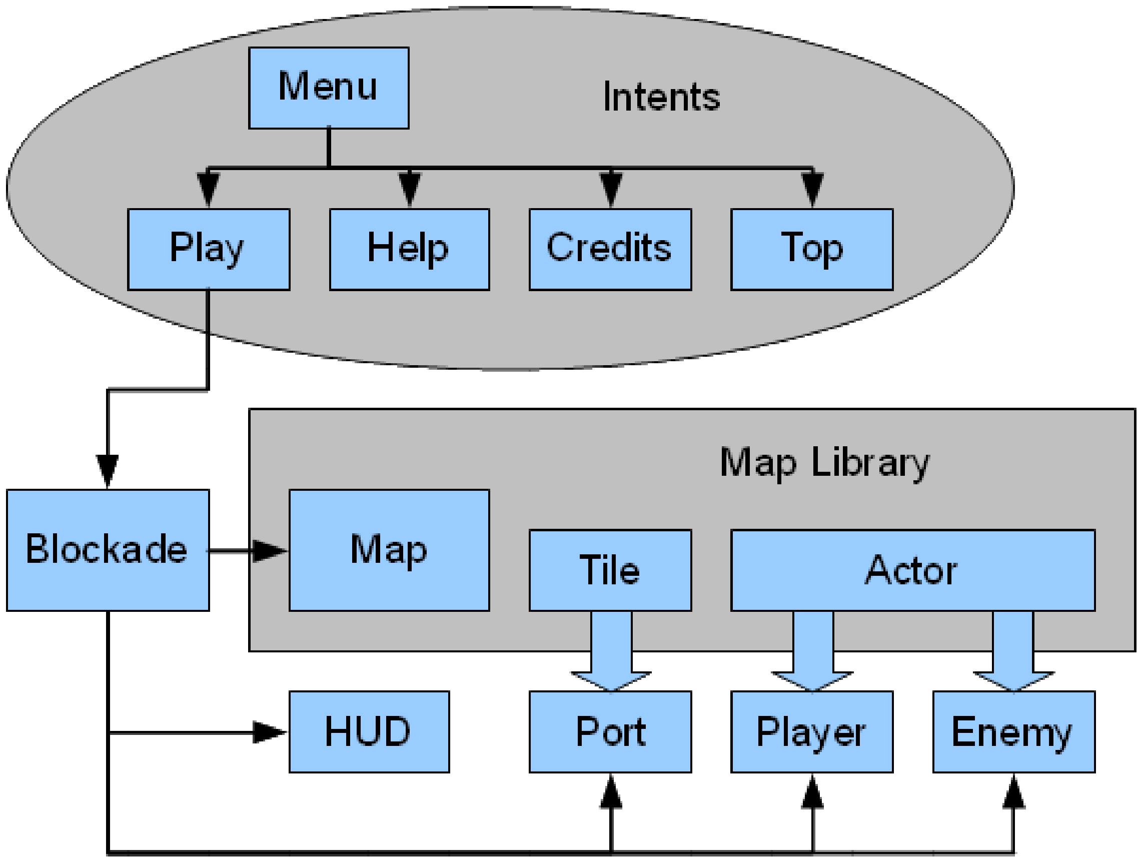

Figure 1: Class Structure. Block Arrows indicate inheritance.

The Port, Player, and Enemy classes all extend a class in the pseudo library, but the Map class can only access them as Tiles or Actors. The Blockade class must maintain references to them, so that it may execute the game logic.

3.2 Art

All of the art assets for Blockade where hand drawn using a pen and ink. The intention was to give the game a unique look, but this method also alleviates any potential copy write issues.

Once the drawings where scanned, they would be resized and then set into a rectangle tile. A hex shaped mask was then applied, and all areas of the rectangle that were not covered by the mask where made transparent. Thus the tiles appear to be hexagons, when in fact they are just offset rectangles.

Early on the decision was made to use a hex tile size of 64 pixels by 72 pixels. Due to the time involved in resizing and aligning the original drawings, this soon became a fixed requirement, as a change would require to much work.

4 Implementation

Overall, the implementation went pretty smoothly. The game works, and it is playable. It is not very entertaining and limited to only one level, but given my lack of experience with Android and the time frame this is expected.

4.1 Problems

I ran into several road blocks while building this game. Most were simply a result of using a new framework and my unfamiliarity with it. For example, I had to learn how to launch threads and deal with synchronization withing Android. Others were the result of the limited time frame, such the omission of sounds from the game. Others are actually slightly interesting, and those I will discuss here.

The decision to use hexagon tiles instead of square tiles proved to be a larger complication then I original thought. Offsetting the tiles correctly was not difficult, but reversing this offset for the purpose of moving a ship was a pain. Since every other row is shifted down by half a tile, each tile has two tiles to its left and two tiles to its right. The row index of these tiles, however, can be one of three possibilities, depending on if the column is even or odd.

This in itself isn't difficult to solve, but it necessitates some ugly code in certain parts, such as the collision avoidance. When the player selects a tile to sail to, there is often the case where the ship must sail around a piece of land. I did not implement any path finding algorithms, so if the land is large enough the player is left to direct the ship themselves. However, there is often the case where the land doesn't appear to be in the ships way, the ship must just move adjacent to the land. I had to code specifically for these cases, due to the hexagon offset issue.

An art related issue, aside from the general problems with programmer art, was that I could not for the life of me draw a ship sailing downwards, to the bottom of the screen. This was remedied by making it so that the ships do not sail directly downwards, but must "tack" to either side. The justification is thus that the wind is "blowing" from the bottom of the screen, so the ships can not sail directly into it.

4.2 Clever Bits

The only pieces of code in the game that can be considered clever are a result of writing general Tile and Actor classes which are then extended as needed. Actually, Actor extends Tile. That's probably the most clever bit.

Tile represents a single hexagonal tile, and is responsible for drawing it and updating it's animation. Actor is basically a Tile that moves, and doesn't cycle animations (since they must be set depending on direction of travel. The Actors class also deals with all of the tricky navigation code to avoid landfall, so I only had to write this once since the Enemy and Player class both inherit it.

However, this caused a problem on instantiation: the Map class only knows tiles and actors. The Blockade class could create the players, ports, and enemies itself, but then it would have to know how to create tiles and actors as well. To fix this problem, the Blockade class parses the level file and calls the appropriate Map methods for creating the tiles and actors. In this call, the final class of the tile or actor (such as Player) is passed, so that Map can dynamically load it's constructor (which by convention are all identical). Blockade can then configure the specialized classes as necessary.

The end result is that Map, Tile, and Actor are all reusable. Even their bitmaps are defined in Blockade.

5 Screen Shots

I have included a screen shot of the game's main menu (Figure 2), help menu (Figure 3), and game play (Figure 4).

6 Conclusion

Blockade! is a decent proof of concept, and I intend to continue work on it. The end goal is a to release it on the Android market, but the level of polish and quality will take time and work.

Regardless of whether or not I continue development, I have learned a great deal from the project.

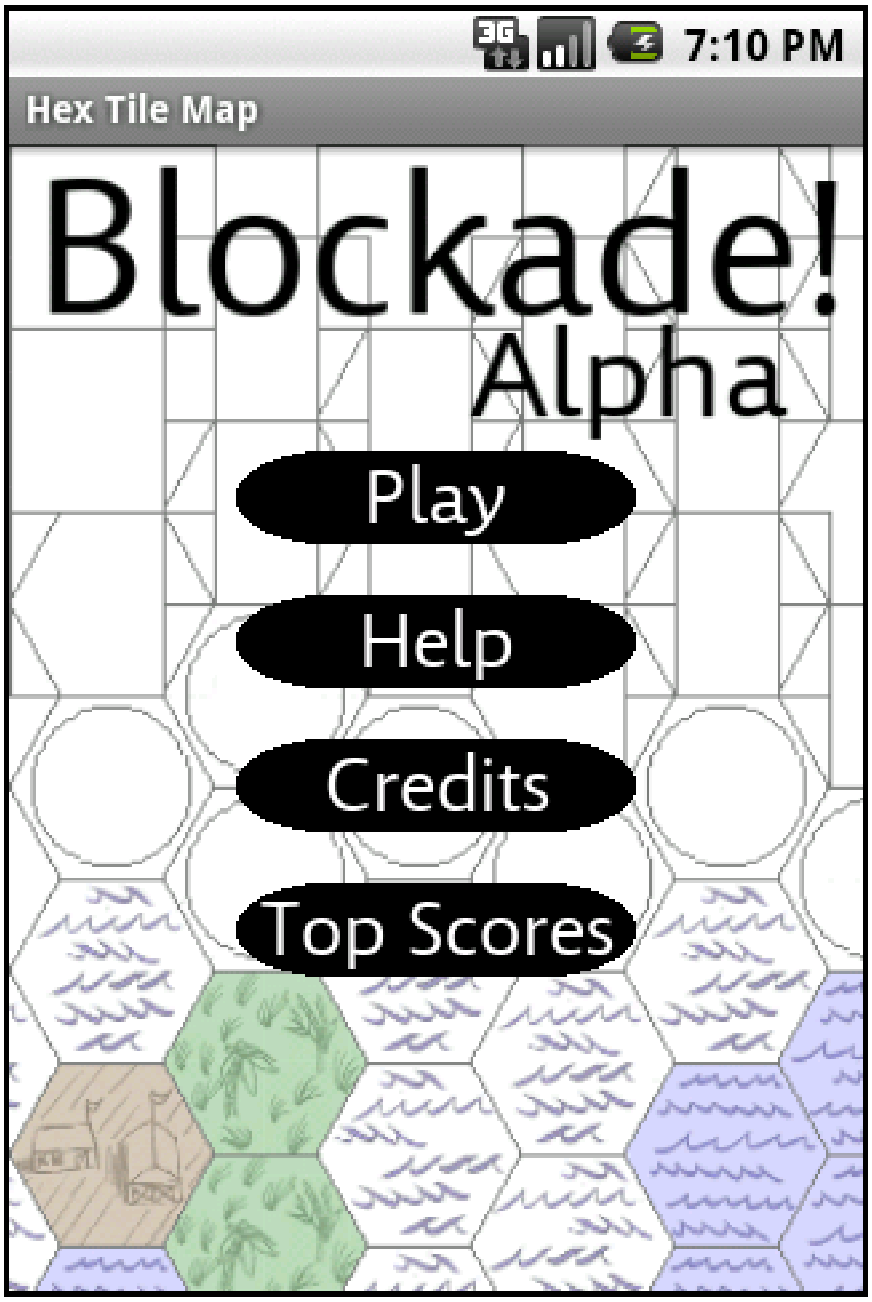

Figure 2: Main Menu

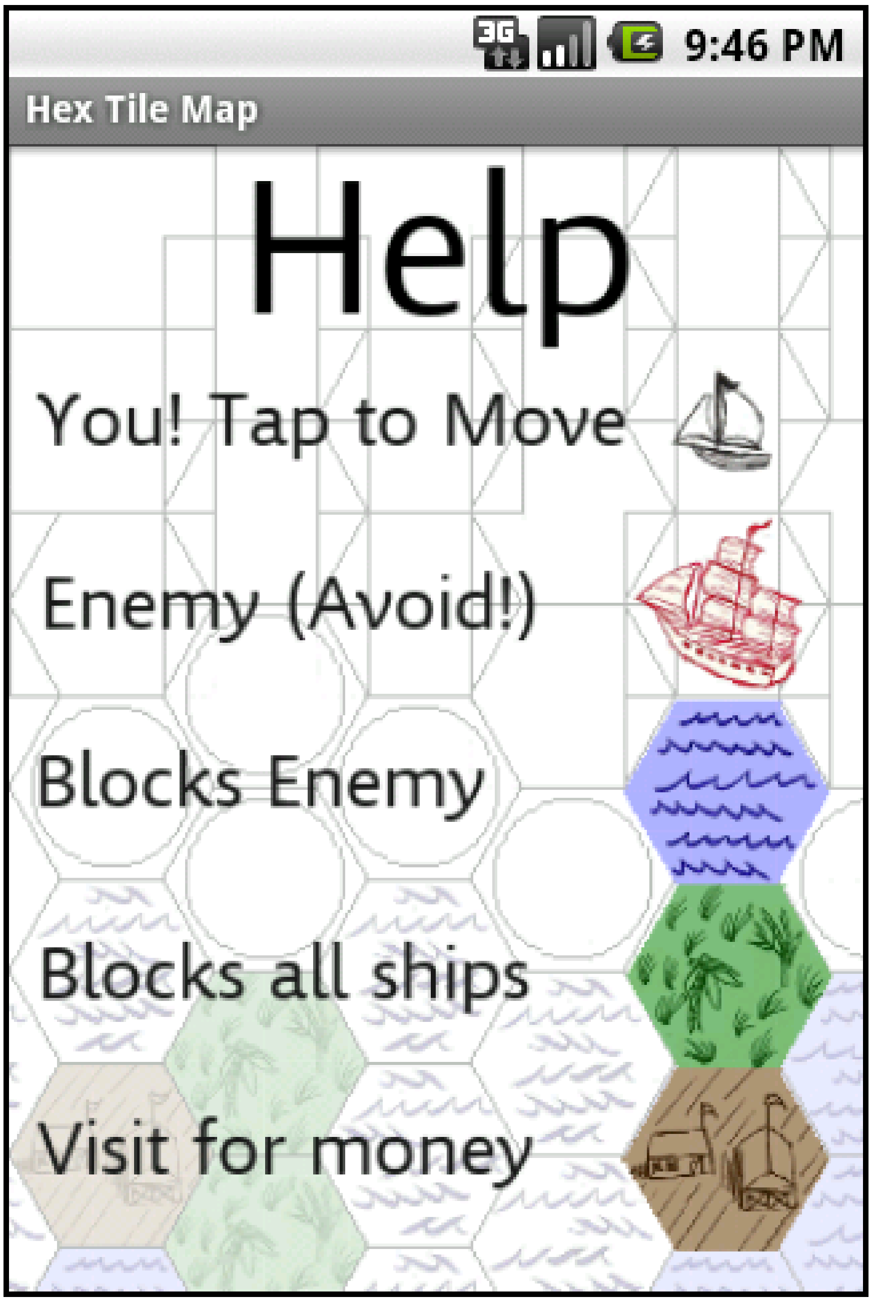

Figure 3: Help Screen

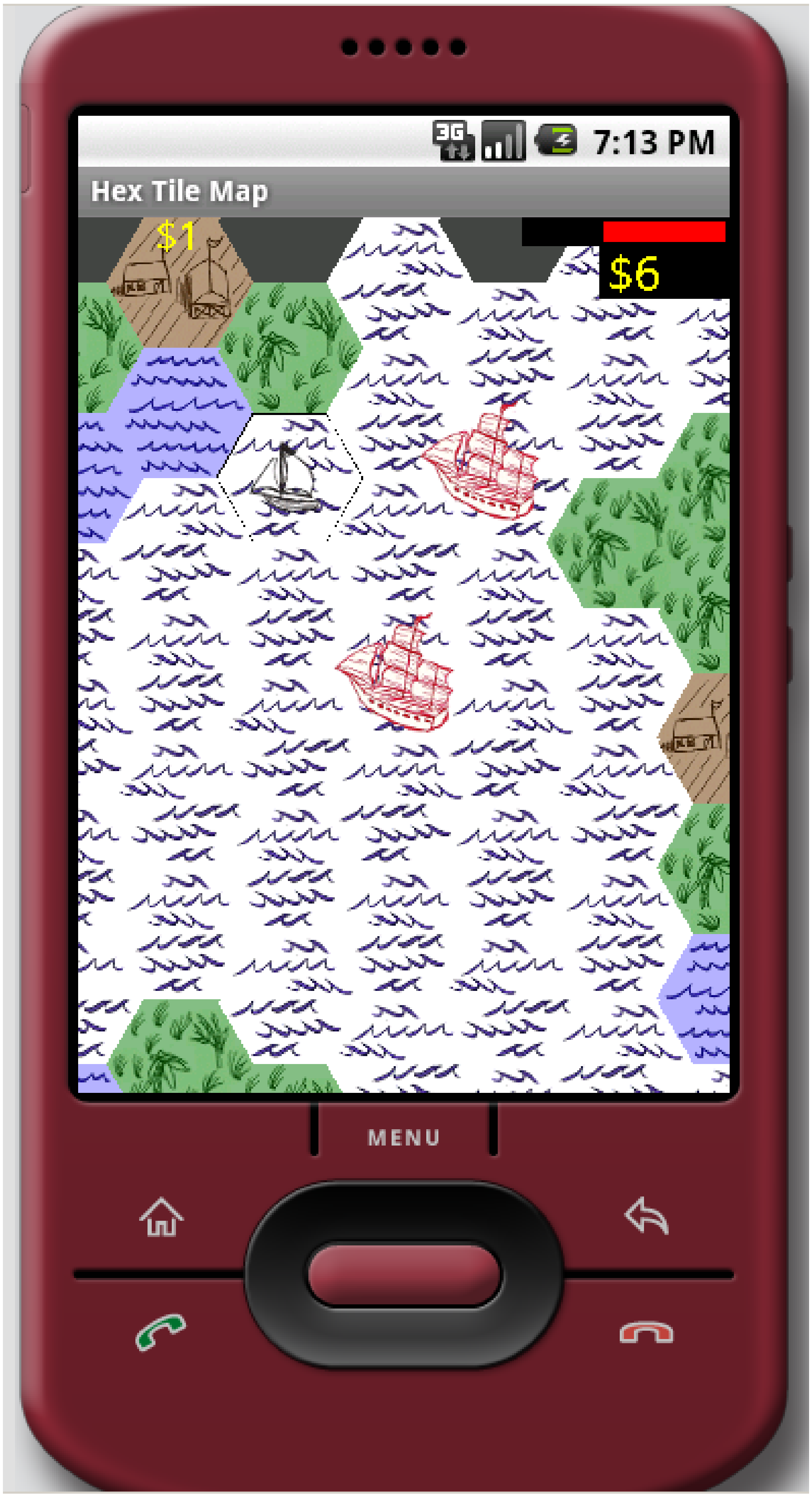

Figure 4: Game play showing phone