Odyssey Now! Magnavox Odyssey Emulation Project

Contents

[Original Project 1](#_Toc500799627)

[Issues Tackled this Semester 2](#_Toc500799628)

[Running the Program 2](#_Toc500799629)

[Analog Slowdown 2](#_Toc500799630)

[Image Flickering 3](#_Toc500799631)

[Diagonal Stepping 3](#_Toc500799632)

[Data On Update 3](#_Toc500799633)

[Jitter 3](#_Toc500799634)

[Port to Mac 4](#_Toc500799635)

[Universal Controller 4](#_Toc500799636)

[Inertia 4](#_Toc500799637)

[Android Port 4](#_Toc500799638)

[Known Issues 5](#_Toc500799639)

[Inertia Bugs 5](#_Toc500799640)

[Collision Detection Failing 5](#_Toc500799641)

[Card 6 5](#_Toc500799642)

[Moving Forwards 5](#_Toc500799643)

[Fixing Above Issues 5](#_Toc500799644)

[Linux Port 5](#_Toc500799645)

[Full Screen 6](#_Toc500799646)

[Comparing Behavior 6](#_Toc500799647)

[Implementing Bugs 6](#_Toc500799648)

## Original Project

The first version of the Odyssey Now! project was started by Pat Healy under the guidance of Professor Zach Horton to emulate the experience of playing the original Magnavox Odyssey. The project is written in Java, and included 7 Java files, 15 Card files, and one file that explained how the cards were programmed.

The project was intended to communicate with a specially designed Odyssey Emulator Controller, 3D-printed and constructed by students. In order to do this, the project uses jSSC, Java Simple Serial Connector, to take the controllers input, parse it, and pass it to the respective classes.

## Issues Tackled this Semester

### Running the Program

For some reason, when the project was first inherited, it wouldn’t run. The classpath to the jSSC files had to be specified, and ultimately I had to redownload the jSSC files to get it to work properly, as jssc.jar rather than just code.

To compile and run this, the class path has to specified on the command line level:

javac -cp "./jssc.jar;" ODSYRunner.java

java -cp "./jssc.jar;" ODSYRunner

Alternately, for Windows based users, two batch files were created to automate this process, BUILD.bat and LAUNCH.bat appropriately.

These files in particular turn the program into a jar itself.

Many operating systems will allow a user to click and run this directly, though it may cause issues on Mac. Unfortunately, the ODSY.jar is dependent on the Manifest.txt and jssc.jar, so we are not capable of distributing the ODSY.jar on its own.

To run the program on Mac, it’s very similar to Windows, but Macintosh specifies classpath differently. It is recommended that this is researched, as I cannot confirm anything I put here would be correct without a Mac to test on.

Be warned, however, that the command line will not give a warning if the classpath is specified incorrectly. If the program does not run, it’s likely the classpath was specified incorrectly. The program was tested on the Macs in the DMAP lab, so at the very least it should run on those computers.

This undertaking also required rearranging the file structure, which is not described in depth, here. Basically, all code was moved to a special folder, along with the card files, which allows the Jar compression to work.

One may realize that the jssc.jar file exists in both the main directory and the ODSY directory. This is necessary for the program to work, and no way around it was found.

### Analog Slowdown

The biggest issue that needed solved was an issue where connecting the analog controller would cause lag virtually to the point of unplayablity. The first goal of this semester was fixing this problem.

Many solutions were considered, ranging from replacing jSSC to optimizing the rate of draw.

Ultimately, the implemented solution ended up being introducing multi-threading.

Multi-threading was something that Pat had attempted to implement originally, but for whatever reason the attempted implementation failed.

The current implementation currently runs on three threads.

The main thread focuses on handling the game logic and moving the boxes and balls.

The thread called GraphicsThreading (I cannot remember why I thought naming it this would be a good idea, I apologize) handles getting the input from the controller and setting the values internally.

The thread called PaintThread handles repainting the screen, drawing the background, the players, and whatever accessories are called for on the card.

This solution was implemented because in the original design, the program was only allowed to update based on the controller in a sequence, only after all other things and data was updated. This resulted in the positions and the screen only being updated at certain times.

After implementing the multithreading, the program runs much smoothly, but introduced the next issue.

### Image Flickering

A digital computer is so much faster than an analog computer, that this solution resulted in the square ‘teleporting’ across the screen if the user flicked the control too quickly.

In order to fix this, new variables were added to the PlayerBox class, called ‘drawx’ and ‘drawy’ that distinguish between where the player is currently being drawn at, and where they want to be at. Each time the screen is drawn, the box is incrementally drawn closer to their target.

This fixed the flickering, but once again, introduced a new issue.

### Diagonal Stepping

When the player would try to move their player in a diagonal direction (as players are wont to do while playing games) the program would try to take a 1/1 diagonal towards it, then continue on the axis that was left. This created a very unnatural and obvious pattern on screen.

In order to fix this, a queue was added to the PlayerBox class, that keeps track of the list of locations that the player has moved to. Now, the program continues incrementally moving to the target, but it now has a list of destinations it needs to make it to.

The user is now free to twirl and the program will draw their box accordingly.

### Data On Update

One previous idea for tackling the slowdown was to modify the Arduino code to only send data if the status of the controller has been changed. We had hoped that this would result in less updates and make the program more responsive. Unfortunately, this ended up not helping the problem (the multithreading was the ultimate solution) but the code remains modified.

When we first made the modification, it ended up being useless, because natural fluctuation in the controller resulted in it updating even when the controller wasn’t being touched.

### Jitter

Even though the code ultimately wasn’t helpful for what we were looking for originally, we did end up implementing code so that the Arduino will only update if there’s a 3-degree difference on the controller.

Originally, if you watched the screen and didn’t touch the controller, you could see the players move of their own accord, jittering slightly left to right or up and down. Implementing the 3-degree moderation helps minimize this.

### Port to Mac

The next project was porting the code to Mac. Due to the similarity of the Operating Systems and the universality of Java, it wasn’t much of a hassle.

However, Windows and Mac (and a useful note for Linux, later) name their Serial Ports differently. On Windows, the controller was named COM4 or COM3, while on the Mac used for testing, it was named something along the lines of “dtl0/usbmodem12345”

Other than this, the code worked fine on both systems. Our next goal became modifying the code to work universally.

See the section above for instructions on how to get the code to work on Mac.

### Universal Controller

Our first thought to solving the problem was to update the controller with a name. However, while Arduino permits this, most operating systems decide to ignore it and assign a name to it themselves. Another possibility was assigning a serial number or such to the controller, but operating systems were inclined to ignore that, as well.

Our solution ended up being a clunky, but effective programmatic approach.

When searching for the controller, it will check all Serial devices attached to the computer, then regex check their input against the expected pattern of our controller, and if it matches, is determined to be our controller.

If, by chance, a user has another controller attached that sends data in the form of a set of 8 one to three digit numbers comma delineated, the code may fail.

But we figure the odds of that happening are rather small.

### Inertia

The last major goal worked on programmatically was inertia. Pat had some code for it added, but it was insufficient and did not accurately reflect Odyssey behavior.

The current solution implemented is to simply reduce how often the screen calculates new coordinates for the game to draw the characters at. This seems to fairly accurately represent “inertia” how it exists in the game, but rigorous tests have not been done.

The code is also buggy, having some unexpected behavior that is evading scrutiny.

### Android Port

Simultaneously, another member of the project started working on the Android port of the program.

We discussed many ways to go forward with this port, but most Java emulators in Android do not support much of the code used in the project.

This resulted in the group member moving forward with completely redoing the game’s logic in Android Studio.

See that member’s write-up for more information.

## Known Issues

### Inertia Bugs

As mentioned above, the inertia implementation has major problems which need to be solved. The controller is not as responsive when inertia is on for some reason, and after a player leaves the upper left corner, they are prevented from returning to those edges, perpetually locked into being about 50 pixels away from both the top and left edges of the screen.

It is currently undetermined what is causing these bugs, but they need to be found and fixed as soon as possible.

I believe this problem may be solved by correcting the next problem.

### Collision Detection Failing

Due to the way that the new version of drawing the square works, it’s possible that the collision detection might be thrown off, because collision is based on the ‘x’ value, while the character is drawn at the ‘drawx’ position. It may be more accurate to simply get rid of the ‘x’ value altogether and add temporary values when necessary, calculating collisions off the drawn positions instead of the target position.

I believe it may be possible that this bug is impacting the inertia bugs outlined above. The ‘onScreen’ function that prevents the user from sending the box way off screen might be triggering incorrectly, putting a strange cap on where the inertia-driven player is permitted to go.

Regardless of whether or not this bug is causing these problems, it should be fixed as soon as possible.

### Card 6

The original Odyssey only had circuitry for inertia on the second player. The code was updated so that the inertia did not affect the second player. However, Card 6 has inertia and only one player. On the real Odyssey, this player has to be player 2 to account for the inertia, but when there’s one player, the emulator defaults to automatically being player 1.

This should be an easy fix, but is one that hasn’t been implemented yet.

## Moving Forwards

### Fixing Above Issues

Obviously.

### Linux Port

Since the universal code has been handled, and jSSC supposedly handles Linux distributions, the code should work on Linux machines as is. However, no one working on the project had easy access to a Linux machine, and this has never been put to the test.

### Full Screen

Currently, the code will only work in the predefined window size. It would be a nice feature to have it resizable. Pat says that the obvious way of handling this problem (scaling all the objects according to the new size) caused too much lag to be feasible. No other attempt on this task has been made.

### Comparing Behavior

It would be beneficial to rigorously compare the emulator’s behavior with the real Odyssey’s behavior, a task that was unfeasible during this semester due to others needing the device for their projects.

### Implementing Bugs

The Odyssey has a few known bugs whose presence need to be replicated in the emulator.

Typically, when a player hits the reset button on a game with the ball, the “ownership” of the ball switches, switching the ball’s direction. However, if both players push the reset button at the exact same time, they both gain ownership of the ball, resulting in it staying motionless horizontally, but both players being able to influence vertical velocity through English.

Like most older video game systems, when a player is “destroyed” they are not actually removed from the game, but simply set invisible, or painted the same color as the background. This allows games such as Cat and Mouse and Interplanetary Voyage to let players “destroy” each other when one box gets too close to the other. However, due to interference in the signals, if a normal player is too close to a “destroyed” player, the area around the said player is distorted, allowing an astute player to spot their location.

## GitHub Repository

<https://github.com/PatHealy/ODSYredux/tree/Fall_2017>

Please note that the updated code is specifically the Fall\_2017 branch, as the updated code has not been merged into master.