# Odyssey Now! v0.2.3

(AKA arbitrary numbers because I don’t understand how version numbers work.)

0.x.x = beta, unfinished

0.1.x = all work done previously

x.2.x = the work done Fall 2017

x.x.3 = the next meeting number of this semester

The project seems to have come with little documentation from the previous owners, minus a brief guide on how to operate it. This semester, myself, Jessica Mesler, and Scott Crawford were tasked with optimizing and porting the project, along with minor bug fixes as they came up.

Not long after working on the project, I got fed up with navigating the command prompt, and constructed two bat files, BUILD and LAUNCH that, as it stands in this version, will successfully compile, build, and launch the file, via clicking on BUILD twice, then LAUNCH twice. Depending on your windows configuration, double-clicking the new ODSY.jar file may launch as well, but the most surefire way is to click the LAUNCH file. During these activities, you may end up with a Command Prompt window up briefly, AKA, that sketchy black box with white or green letters, and this is perfectly normal.

# Launching Odyssey Now!

1. Double click BUILD.
2. Double click LAUNCH.

If this doesn’t work, you likely don’t have Java installed.

You may get security warnings from Windows saying it blocked the run. I can tell you they’re safe, but of course it’s your choice if you believe it or not.

# Running From the Command Line

I wanted to make sure that developers could still compile and run the code from the command line if they so choose, so I ensured this compatibility remained inside the code, but due to the JSSC problem outlined below, you’ll need to explicitly state the dependency during compilation and running, in the style of:

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

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

# TO DO:

The compatibility of Odyssey Now! with Mac and Linux is under evaluation. Ideally, this project should run on all three operating systems.

# Modifications from v0.1

CardReader

In order to get the BUILD and LAUNCH files to work, I needed to modify the CardReader class. This only consisted of pointing it in the right direction if it can’t find the Card files.

The way it currently works is catching the exception, trying a different location, then throwing a new exception if it still can’t find it. I don’t entirely like it as it seems inefficient, and I’ll try to find a different way to go about it.

I didn’t want to eliminate compatibility with the command line, so the program attempts to look in both locations before giving up.

JSSC

For an unknown reason, the current build on GitHub refused to run on both new team member’s computers. To solve this problem, we found a JAR build of JSSC online, which solved the dependency problem. Since this file contained many of the same files in the build previously, those files were deleted in favor of the working JAR.

AnalogInputListener

An experimental class that exists, but currently doesn’t actually modify the operations of the program. Theoretically, this class would split into 6 (or 8? It was unclear by the code) with each one listening for a specific input. I.e. thread 1 would listen for a change in xPosition, thread 2 would listen for yPosition, etc. However, as the specific operations of the microprocessor and JSSC are currently unknown to me, it’s uncertain if this idea would work.

# Optimization

The biggest concern for the project currently, and rightfully so: who wants to play a game that lags?

Analog Input

The analog input is suspected to be the cause of the lag. I currently have not managed to witness it personally, but I have a theory as to what the problem is.

A processor operates very quickly: millions if not billions of instructions per second. Of course, not all of this is dedicated to our program, and there’s the issue of translating from high level to low level, but it’s still a very quick operation.

The way that this program is running is referred to as “polling.” It means that occasionally the program stops and asks the controller “Hey, what state are you in?”

However, due to the speed of the computer, it’s entirely possible that this polling is happening *quicker than we can physically move the controller.* Even if it isn’t, there’s still a lot of blank time when we’re thinking (much, much slower than a computer is) or just strategically staying still and the program is wasting time fetching values that haven’t changed.

The program has a fetch rate built in, but then you run into a separate problem. Set the fetch rate too low and you lose optimization, and set it too high and you lose reactivity.

That’s why, ideally, we would want something like this to be an interrupt driven program. This means that the program spends most of its time calculating and drawing, but when a player interacts with the controller, the controller notifies the program (called an interrupt) of the change, and the program reacts accordingly. Therefore, the program only has to worry about interacting with the relatively slow medium of the controller when it’s necessary, and also doesn’t have to worry about missing interactions.

How to achieve this is the complicated part.

While speaking with the professor, he mentioned that there may have been a setting in the microprocessor to only send data when it has changed, and I highly encourage looking into this. While in the discussion with the professor, I was saying that we would still need to modify the code to handle this system, and he was asking why we would. After looking over the code, I understand what he was trying to say, and believe it comes down to the implementation of JSSC. The receiving of data is monitored by the readString command in their implementation: if this is a blocking call, I believe this may be a very viable option. If not, further work may be necessary, which I suspect involves coding device drivers and working with the AnalogInputListener that I started work on.

The previous team went with the current option as it seemed to be the better approximation of an analog system—and the goal of the project is to approximate the analog as closely as possible on a digital system, but I don’t think there’s anything to be concerned about. It will still be a continuous, analog system, but the duplicate values will just be filtered out. While these duplicate values might be important in some circumstances, this is a digital emulation of an analog system where the values would just be discarded anyways.

Reduce Repainting

A much less viable option than figuring out why so much slowdown occurs with the analog, but one that will add the slightest bit of optimization in exchange for some memory.

Essentially, every time the game refreshes, it repaints the whole screen. Some time can be saved by remembering previous positions and only repainting what is necessary. The optimization from this endeavor is probably minimal at best, however, and may or may not be worth the work that would go into it.

Miscellaneous Optimization

The Java compiler is young compared to some other compilers (C, looking at you) and as such sometimes you can squeeze extra performance out of it. Loop unfurling is one example, along with ensuring that variables aren’t created and discarded recklessly. I would also be interested in keeping the serial port open continually instead of repeatedly opening and closing it. Again, I don’t think any of these will help as much as getting the analog input sorted out.

# Porting

Theoretically, JAR files should work across all platforms—it’s the benefit of using a cross-platform language. However, there are upsides and downsides to using them, and it will need to be discussed later. First, testing needs to be done to ensure the program runs on all three systems.