Begin: 3/11

**6 November 2016**

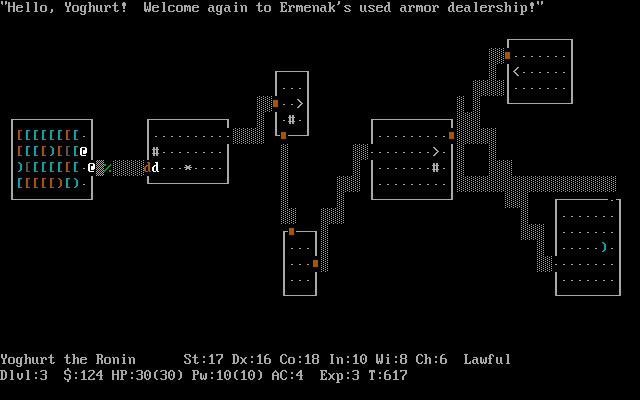
Begin project, the assignment was received on 3/11/16. Will begin considering ideas soon.

**20 November 2016**

Continued thinking about concepts for the game. I came up with a few ideas.

1. A 2D rogue-like

These games allow the player to control a person which moves around the dungeon, fighting monsters and attempting to go as far into the dungeon as possible. It would be quite easy to fulfil the requirements, and would be interesting. There are several good examples of these games made before, e.g. NetHack



NetHack, a very old rogue-like

1. A programming game

In this, the player need to write code (possibly in a fictional language) to create algorithms to solve puzzles. Zachtronics has made a few of these games (e.g. TIS-100), and this kind of game would be interesting to make and play. However, it does not have as distinct ‘lose’ condition (though this may be discussed).



TIS-100, a recent assembly programming game

1. A game similar to ‘gyro’

Gyro is a mobile game which is quite old (2 years), where you control a central circle with different coloured sectors. The player needs to rotate it so incoming balls land in the correct coloured area. I quite enjoyed playing this game before, but it no longer works, so I want to remake it. It is also a simple concept,



Gyro Gameplay

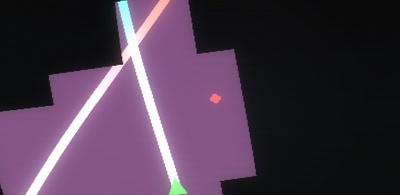
1. Orbital Mechanics

The main idea behind this game is to use gravity in space to achieve some goal. I have done something similar before (using JavaScript), but the game will need to be much more developed.

*[ no image available ]*

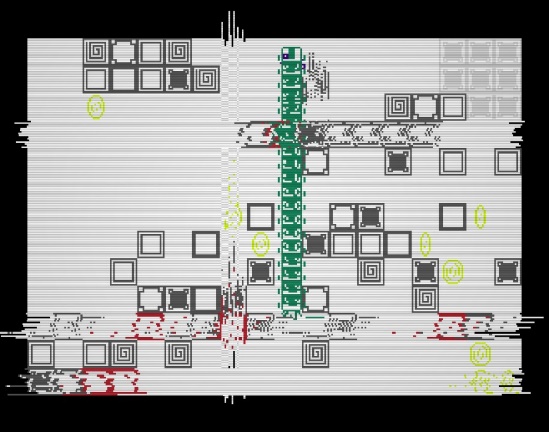
**4 December 2016**

I found a game that would likely be better to base off. It is called “jet/lag” (<https://svblm.itch.io/jet-lag>), and has fast-paced gameplay similar to other games I like. As a result, the game would probably be based off this.



Jet/lag gameplay. It’s hard to show in an image

Another game that would inspire the project is “glitchhiker” (<http://www.glitchhiker.com>).



*GlitchHiker, rather chaotic*

I will probably base the final game off these, with modification of course.

After thinking about this idea, I worked on the documentation. Specifically, the Gantt chart.

**18 December** **2016**

This is the first week of the holidays. I looked at various frameworks and libraries that could be used for the engine. I plan on using SFML for input/output (inc. display & sound). I previously also have made a vector library which I plan on using (unless another math library succeeds it). An entity system will also be needed – EntityX or anax will be used (though not currently decided).

During the second week, I went to Melbourne. As a result, I was unable to work on the project.

**1 January 2016**

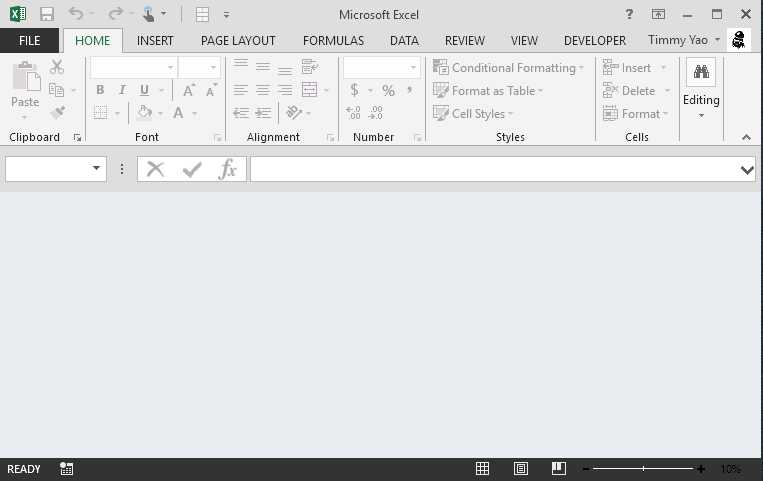
During the first week, I made a git repo for the project, which I will use to store the files created for the project (e.g. documentation, source code). I move the documentation over to that location, but did not move the source code (which stayed as a folder simply named ‘test’ on my desktop).

I experimented with using SFML, since I haven’t used it in a while. I found a decent vector library named CxxSwizzle (<https://github.com/gwiazdorrr/CxxSwizzle>), which will replace my own library. I didn’t try using either entity system yet – I don’t have the framework (e.g. a graphics system) necessary to use them.

**15 January 2017**

I made a simple implementation of signals and slots, which used a priority queue. Using this, I developed a few utilities to help with developing the game (runtime.cpp). I then continued working on the Gantt chart.

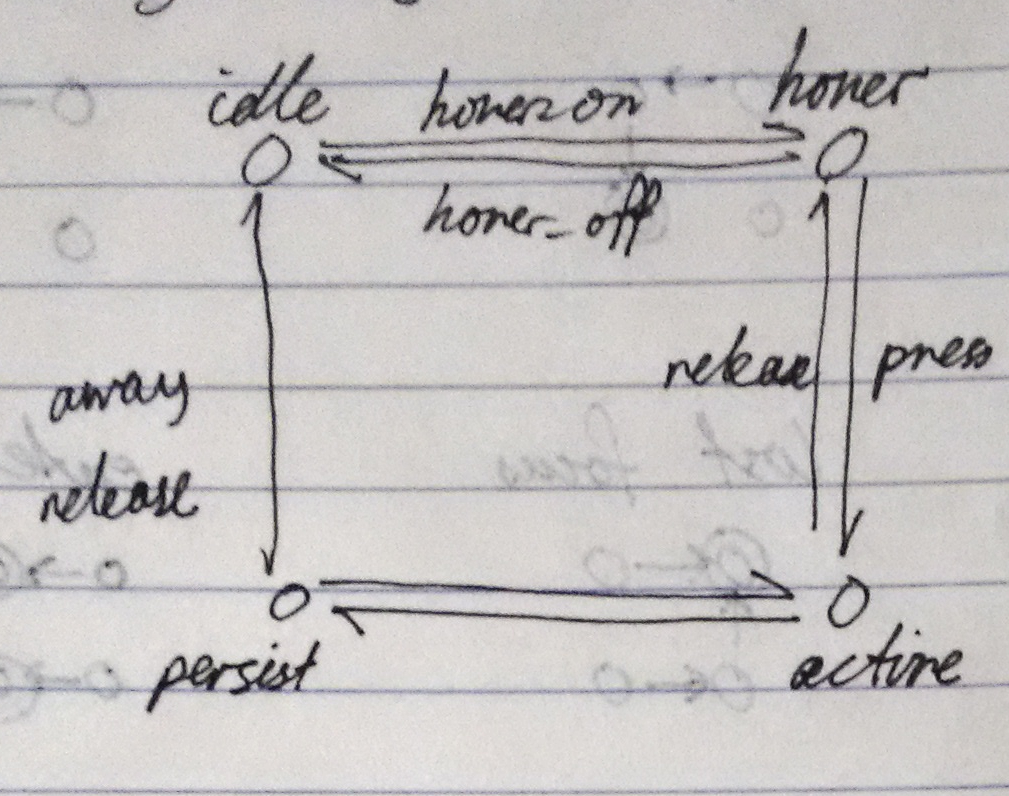
While working on the Gantt chart, Excel suddenly crashed, leaving the chart corrupted. Following this, Excel reported that closing another file I had open would close the corrupted file (?!). Even more surprising was that, instead of offering the option to recover, I was presented with a blank screen (shown below). I was finally able to recover after explicitly telling Excel to try to recover the data (press the arrow next to “open” in the open file selection dialog).



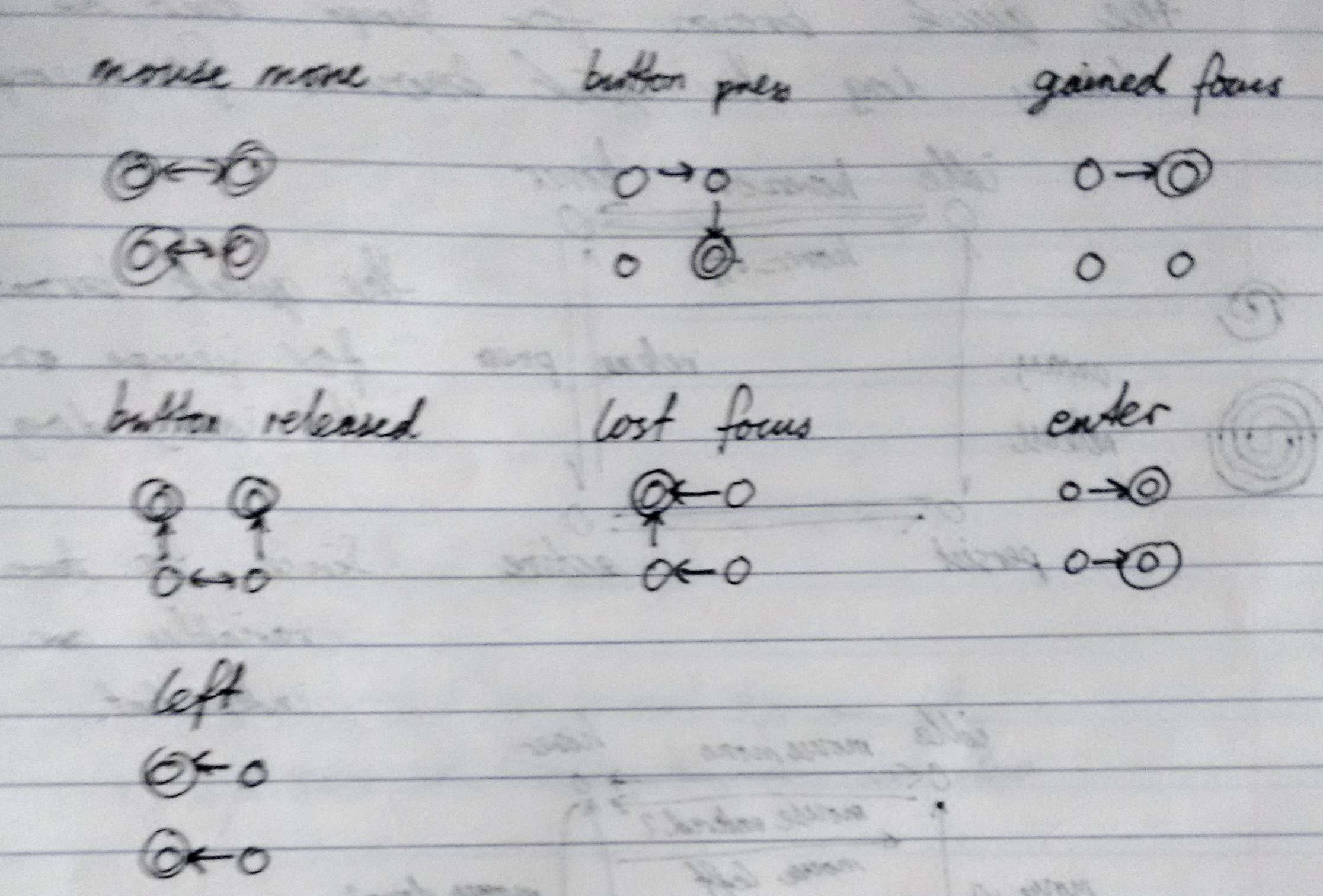
*A broken and confused Excel*

This broken file is still available as “broken-Gantt-original”, unless I make changes later. After this incident, I paused progress on the Gantt chart, and continued working on the code.

I finally moved the source code over to the repo. For the next few days, I worked on a button system for easily making buttons. I soon discovered that it was effectively a finite state machine, and spent about a day figuring out the transitions (no code yet).



*State representation, and transition names*

**

*Transitions when reacting to events*

From this, I implemented the button manager (as input/button.cpp). Using a test program, I verified that I implemented the machine correctly (it was). I then went back to working on the Gantt chart, and finished it.

**29 January 2017**

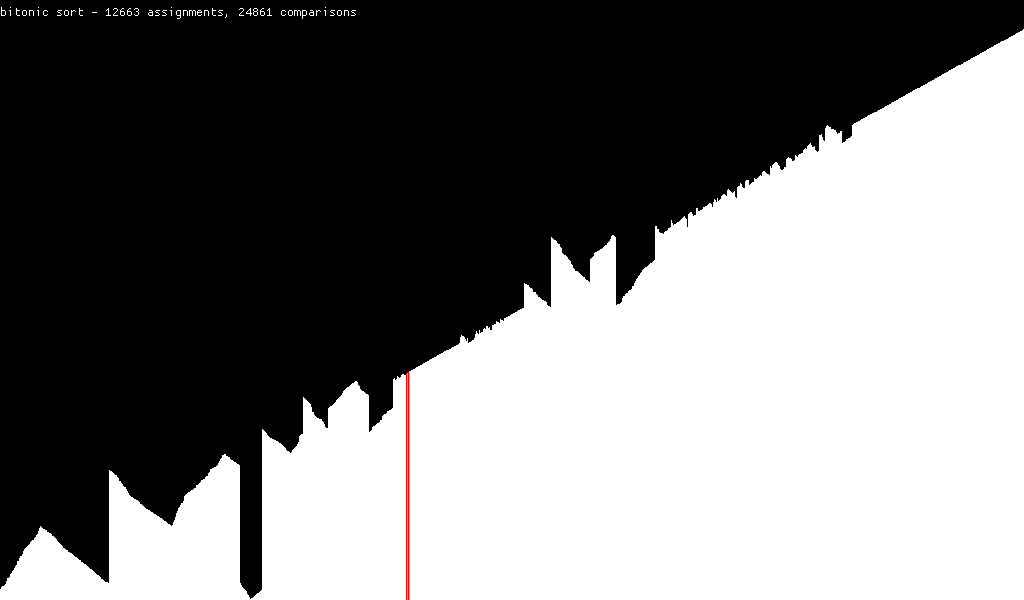
I got a new laptop (yay)! This meant moving my files from the old one to the new one, and installing various programs and such. This was rather time consuming, and the lack of a programming environment effectively stopped progress (I didn’t want issues from working on the project on 2 computers).

However, I did eventually set it up and could resume programming. I began implementing the necessary components to enable CMake to build the project. This would make compiling and testing programs much easier.

**12 February 2017**

I made a sorting algorithm visualiser as an “example” project. I got a bit carried away with it, but it looks great. From this, I learned the capabilities and limitations of the framework:

1. I needed to suppress warnings from external libraries (e.g. CxxSwizzle, which had several). These warnings made the compiler output too verbose, obscuring actual issues with my code.
2. There needed to be a way to convert from CxxSwizzle vectors to the SFML vectors. In my code, I mainly used CxxSwizzle vectors, as they have a better API. SFML API’s, however, require sf::Vector2f, which are not implicitly convertible from CxxSwizzle vectors. Since both are external libraries, there is no easy fix. The façade design pattern could be used.
3. RandUtils is a great library for random number generation.
4. The event-driven runtime encouraged modularisation. This made the program easier to implement (as separate modules) and made it more maintainable (since each module is independent).
5. Resource loading was annoying. Since the location of the executables is different from that of the resources (e.g. fonts, images), it was difficult to ensure that they were always accessible. I resorted to hard-coding them as byte arrays (since SFML can load files from memory), but these are nasty to create and maintain.



*Later stages of Bitonic sort.*

I also added a command line option parser (core/opts.cpp) to allow easy modification of some internal settings (e.g. window size and framerate). Since the sorting algorithm performed one action per frame, this was quite beneficial to ensure the slower sorts (e.g. bubble sort) actually finished in a reasonable amount of time (still a few minutes).

**26 February 2017**

Looking back on my prelim SDD game, I found that I should separate the implementation (.cpp files) from the interface (.hpp files). Then, the runtime library can be compiled separately and only once, improving build speed.

I fixed issue #1 by adding code to the wrapper headers to suppress these errors. Since the external libraries are only accessed through these headers, this fix was simple. See include/vector.hpp, include/randutils.hpp, etc.

I also began creating a solution to resource loading (issue 5 above). I figured that I could create a consistent interface for both on-disk file resources and hard-coded ones by having both accessible from memory. Hard-coded resources are simple, but on-disk resources required memory-mapped files (and all the issues of win32api). With a unified interface, I created 2 utilities:

1. A program to create hard-coded resources. This involved writing out the file as a byte array, and adding the headings to make it valid C++.
2. A program to generate the resource interfaces. Whether on-disk files or hard-coded data was used was decided by a switch, and the program generated the header and source files from the names of the files.

After adding CMake support, this was much better than before.

I also started experimenting with EntityX, the entity component system library I was going to use. After seeing that anax had a critical bug (see <http://tilemapkit.com/2015/10/entity-component-systems-compared-benchmarked-entityx-anax-artemis/>), I decided to use EntityX.

After some fiddling with CMake to build EntityX correctly, I created a starfield example. It was much simpler than the one without EntityX (1.5 screens vs 2.5 screens), and permitted running at any framerate. The lack of dependence on the framerate meant that I could run without FPS limiting, and got over 2000 fps (but it stressed the CPU quite a bit). Hilariously, it also works at extremely low framerates (like 1fps), and appears as if you have a bad computer.

**12 March 2017**

HSC exams are coming up, I’ll be taking a break