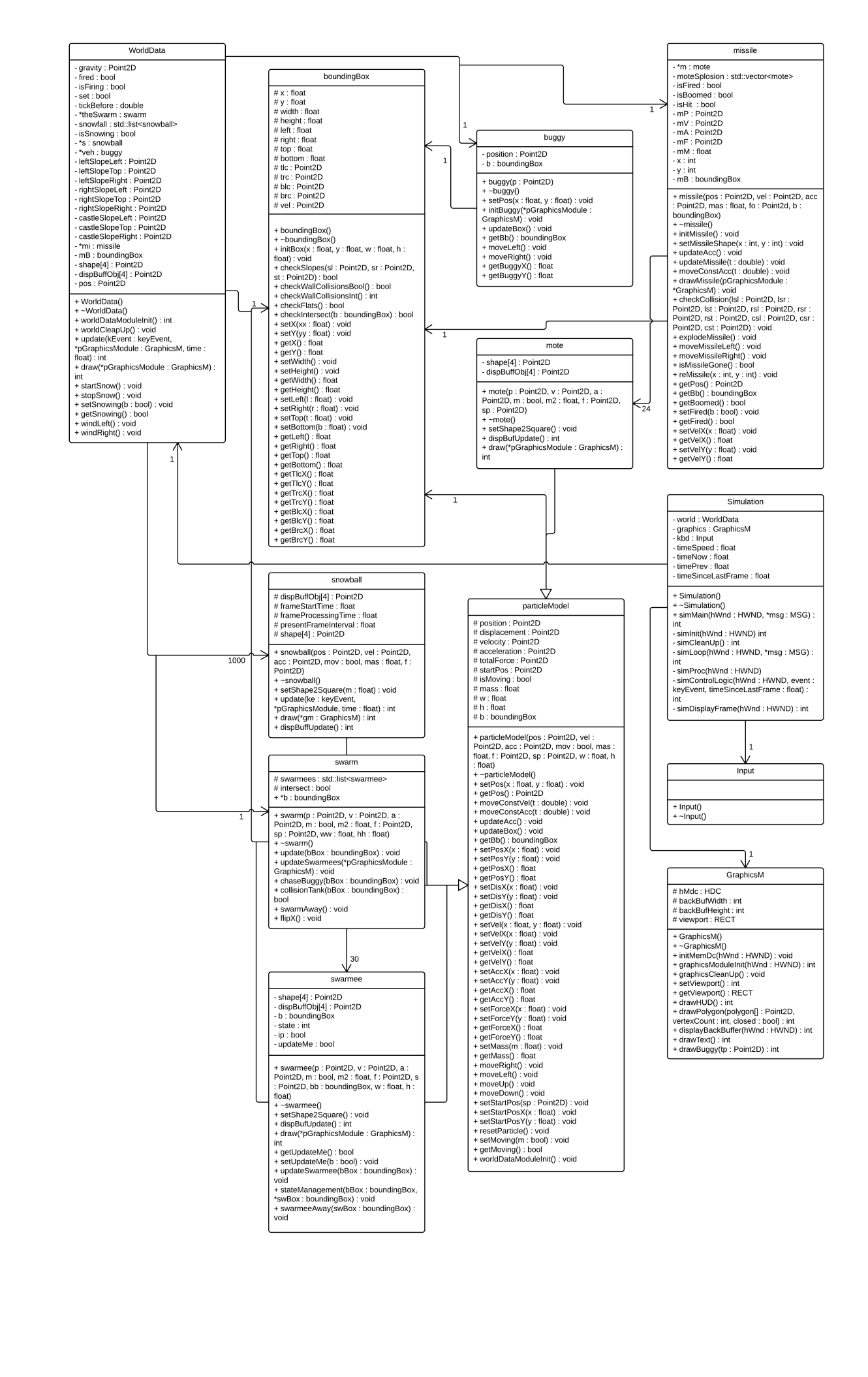
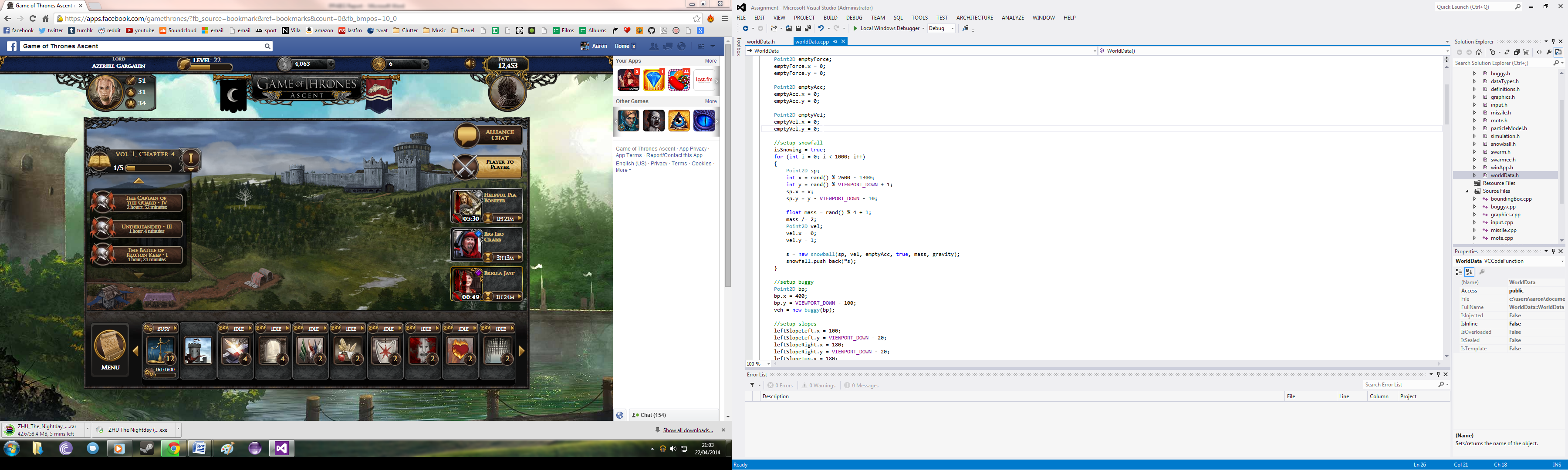
PPAIEG Report - Aaron Austin-Lacy

This module's assignment's intention was to create a small game with a given scenario, whilst implementing certain physics and artificial intelligence aspects to make the game engine more real. After working through tutorial work I then started a new project and implemented some of the new methods I had learnt. My final product was an unfinished program which does not contain the game elements, but has some implementation of physics and AI.

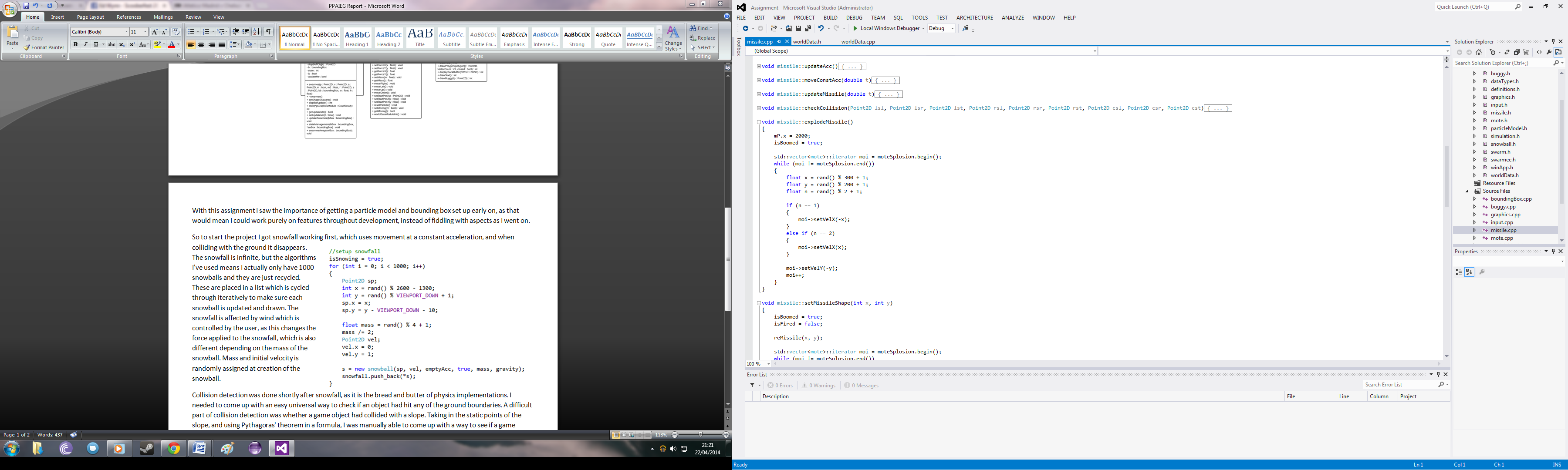
This is the UML Class Diagram of my final program: (larger version included in .zip file)



With this assignment I saw the importance of getting a particle model and bounding box set up early on, as that would mean I could work purely on features throughout development, instead of fiddling with aspects as I went on.

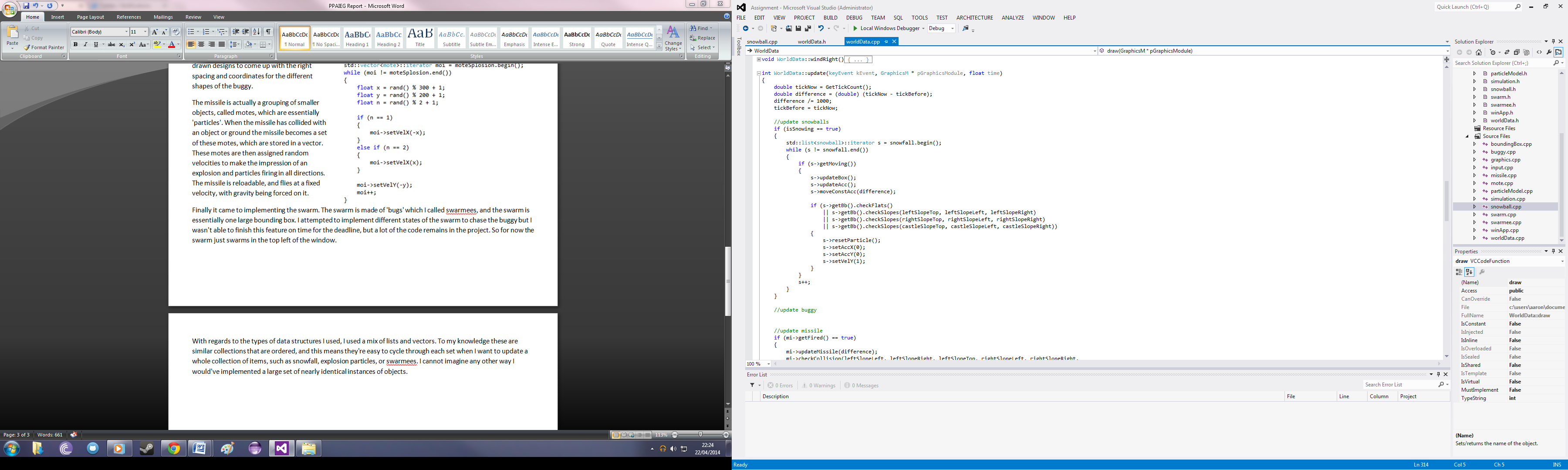
So to start the project I got snowfall working first, which uses movement at a constant acceleration, and when colliding with the ground it disappears. The snowfall is infinite, but the algorithms I've used means I actually only have 1000 snowballs and they are just recycled. These are placed in a list which is cycled through iteratively to make sure each snowball is updated and drawn. The snowfall is affected by wind which is controlled by the user, as this changes the force applied to the snowfall, which is also different depending on the mass of the snowball. Mass and initial velocity is randomly assigned at creation of the snowball.

Collision detection was done shortly after snowfall, as it is the bread and butter of physics implementations. I needed to come up with an easy universal way to check if an object had hit any of the ground boundaries. A difficult part of collision detection was whether a game object had collided with a slope. Taking in the static points of the slope, and using Pythagoras' theorem in a formula, I was manually able to come up with a way to see if a game object had collided with a slope. However using this I wasn't able to calculate new velocities for objects, as if there was a bounce impact, so the snowfall just disappears when it makes it to the ground.

Next was creation of the buggy and its missile. The buggy is drawn through the graphics class, and as such is not an actual object. This prohibited me from doing more with it creatively, but I was still able to use it to check for collisions and the like. It took a lot of hand-drawn designs to come up with the right spacing and coordinates for the different shapes of the buggy.

The missile is actually a grouping of smaller objects, called motes, which are essentially 'particles'. When the missile has collided with an object or ground the missile becomes a set of these motes, which are stored in a vector. These motes are then assigned random velocities to make the impression of an explosion and particles firing in all directions. The missile is reloadable, and flies at a fixed velocity, with gravity being forced on it.

Finally it came to implementing the swarm. The swarm is made of 'bugs' which I called swarmees, and the swarm is essentially one large bounding box. I attempted to implement different states of the swarm to chase the buggy but I wasn't able to finish this feature on time for the deadline, but a lot of the code remains in the project. So for now the swarm just swarms in the top left of the window.

With regards to the types of data structures I used, I used a mix of lists and vectors. To my knowledge these are similar collections that are ordered, and this means they're easy to cycle through each set when I want to update a whole collection of items, such as snowfall, explosion particles, or swarmees. I cannot imagine any other way I would've implemented a large set of nearly identical instances of objects.

This code is in my WorldData.cpp's update method, and it shows an iterator through the snowfall list which updates each item inside the list. This is an efficient way of doing so, as I never add or remove items from the collection, I simply reset them to their original position after they've hit the ground.