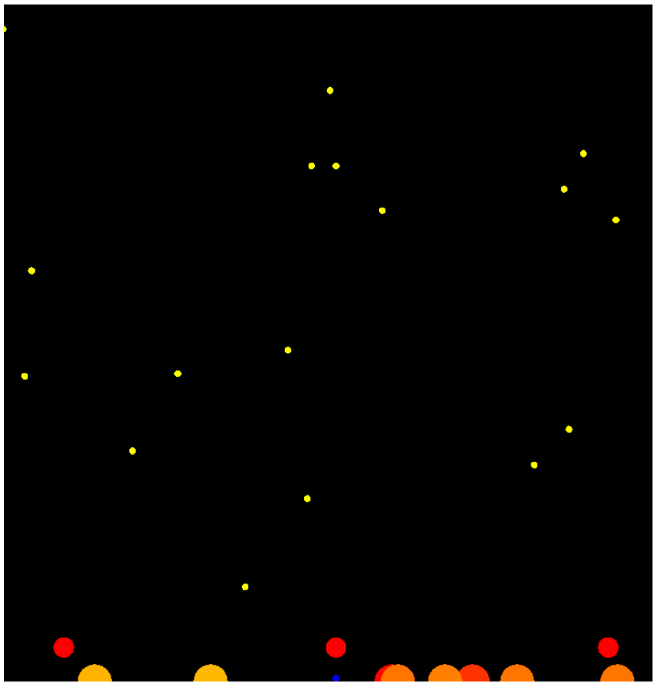
Part 2. Coding Problem (85%)

This portion of the assignment assumes you have some experience in coding C++ so, if you do not, it might be a bit of a challenge and you should start early.

Since you all are in a robotics program, why not have one of the first assignments you do here be programming a robot? More specifically, you will code an AI that will try to survive the longest in a barrage of virtual missiles.



This assignment will walk you through the basics of building a very simple AI that will do very well in simple scenarios (1 - 2 projectiles in the air).

Part 2.1. Setup

This software can be run on any OS, but I will only detail installation for Ubuntu 14.04. Install cmake, build essentials, X11 libraries, and opengl using the commands *sudo apt-get install cmake build-essential xorg-dev libgl1-mesa-dev mesa-utils*. In order to build the program you will need a C++11-compliant compiler; if you are using Ubuntu 14.04, you should have at least gcc 4.84, but you can check the version using *gcc -v.* Next, download the source code from Canvas and extract it. Inside the folder you should have the subdirectories src, include, shaders, etc. Make an additional directory "build" using mkdir. Go into that directory within a terminal and run "cmake ..". We will go over a bit about CMake in class, but feel free to read some tutorials online. This command will set up a build environment inside the directory build. Then, within the build directory, run "make". Your executable should now be built and you can run it with ./missiledefense.

Controls: Press ‘C’ to create an AI controller player. Press ‘Space’ to create a user-controlled player. Use left and right arrow keys to move your player.

Part 2.2. Game Mechanics

Now it is time to dive into some code and understand the exact mechanics behind the game. **You don’t have to write up answers for any questions asked in this section.** This is just an overview of the mechanics and a guide on diving into the details of the mechanics.

As you can guess, the goal of the game is to have your character survive as long as possible. Your character is denoted by the blue circle at the bottom of the window. There are enemies on the map, denoted by the large red circles, which shoot projectiles out of them in an attempt to hit you. These projectiles are yellow and act as simple newtonian objects under gravity. When the projectiles hit the ground they generate a fixed-size explosion that is dangerous for any player to touch and will result in the player losing the game. The only method by which your player can avoid projectiles is by moving horizontally along the ground.

To dive deeper into the mechanics of the game, we are going to explore some of the game code to get a better idea of what’s going on. Take a look at **Game.cpp**, specifically the function tick. This is where all the game update logic occurs.

- If you are not familiar with the structures used in the loops that are in the function, take a look at this tutorial and peruse the documentation for vectors and lists.
- Think about what types the iterator is being dereferenced (“*”) to.

Next, examine the tickProjectile function.

- What is the purpose of the function?
- Why is the Projectile passed with a &? (And what does the & mean? See this.)
- Do the update equations for position and velocity make sense to you?

Feel free to take a similar tour of the remaining functions in Game to familiarize yourself with the game logic. Also take particular note of the order in which things like “explosion checks” are done and when explosions take effect.

Feel free to peruse any of the cpp and header files available. All of the code that you are going to write will be in the Controller class.

Part 2.3 AI Mechanics

The AI that you are going to write is going to be pretty simple. You will be given the whole state of the game (enemies, players, projectiles, and explosions) and you will try to stay alive. The controller class is already scaffolded (i.e., function headers are written) in order to make this straightforward, but feel free to deviate from the design outlined here. In the end the only methods you need to leave alone are the public functions *control* and *createPlayer*.

The most crucial aspect of building this AI is predicting when spots on the ground will be unsafe (due to explosions). Therefore, we need to know where and when projectiles are going to hit the ground. The function in controller responsible for this is trackProjectile. Its task is to return both a time and a position for point of impact. To do this it can use constants taken from the Game object.

The next task will be to actually determine which spots are going to be safe and when. This is the function of the determineSafeSpots method. This function uses the results from trackProjectile to determine which spots on the ground are going to be safe. *HINT* if you inspect the game you will discover that the way the game checks if a certain "spot" is unsafe is by dividing the game space into W cells. When projectiles explode, cells are marked unsafe, and when they disappear, they are marked safe again. It stands to reason that when you are determining when spots are safe/unsafe you might use a similar discretization approach. The function prototype is also up to you and the method of storing information is up to you.

The next helper function is pickSafeSpot. Depending on how you would like to implement the AI, this may or may not be entirely useful, but I found it useful in testing my last two helper functions. Just like it says, pickSafeSpot just picks a spot from the determined spots and returns that.

The magic all happens in the function control. This is where the AI actually gets the chance to move its player. You can test your functions by determining safe spots, picking one, and instantly moving your player there. Doing this means that your character should survive for a long time (but floating point error might cause you some trouble after a while and cause your AI to fail).

In the last test you broke kinematic constraints by moving your player anywhere on the map. In the final version you are going to enforce the constraint that in a given timestep t, you can only move a max distance given by the constant in game (playerSpeed). The exact logic on how to move is up to you.

Grading:

Your AI should be able to predict and avoid projectiles in the "Medium" Scenario **and** survive for at least 1 minute. (Keep in mind this is randomized so the expectation is that your AI will survive longer than 1 minute and that it's not just luck) **Your AI is not required to survive in difficulties Hard and above**; these scenarios are available for you to test and try out. The scenario can be adjusted within the startup.cpp file by changing which "setup" function is used in the main function. We will only accept changes to Controller.cpp and Controller.h. You must also write a readme answering the following questions:

- Write an overview of how your AI works, including how it detects where projectiles will fall and how it chooses where to go.
- What challenges did you face when writing an AI?
- How well does your AI work on a Hard scenario? If it doesn't work, why? If it does, try harder scenarios and see when it does fail and explain why?
- What did you think of the assignment and did it meet its goals? Why or why not?

Write a pdf with answers to the practice problem in the first section named TeamX_AndrewID_Task5.Part1.pdf. Compress the pdf and Controller.h and Controller.cpp and the readme into a tar.gz named TeamX_AndrewID_Task5.Part1.tar.gz and submit it on Canvas.