**Tower Defence Game Design  
Ghusharib Chohan 605**

***Decomposition of the problem:***

Making a game is a very complex procedure that requires pages and pages of code to be made effective. Fortunately, as games are very structured and linear, it is fairly easy to make as you can code parts of the game at the time in a modular approach. The modules can be broken down as follows:

*The game module –* This module is more of a main module where all the code is being run. As the game runs sequentially, I will require the different modules of code to run one after the other in a structured order. I will also be required to find out when the user wants to end the game and as such close the game in a safe manner upon request. All of this should be contained in one module.

*Simulation Module –* One of the modules needs to be dedicated to simulating the running of the animations of the game. This needs to be done in a simulation module as each animation needs to run one after the other. The module will need to get the correct object and change its photo to the next image at the correct time to produce an animation.

*Updating the objects –* Another Module is required to update the values of objects running in the game. For example, when an enemy is attacked, their health needs to go down and be registered as being hit. This will therefore be updated by the module so that the object’s health can be decreased accordingly for every enemy that is hit.

*Updating the currency –* When the user earns in game money, this needs to be registered and outputted as a display to the user so that they know how much money they have available to spend. Equally, the module of code should be able to decrease the amount of money available when it has been spent. Should they not have enough money for a transaction, an error message will need to be shown.

*Updating the score –* Every time an enemy is killed, a running total of the score will need to be updated dependent on the type of enemy killed. This will also then have to update the user interface to show the current score and make the user aware.

*Save high score module –* The game is endless until failure, so a save file is not necessary. However, it would be very useful to have a file with all the high scores contained in it for the user’s reference. This will be required to be updated, added to and outputted on the user screen upon request and therefore requires a module of its own.

*Menu module –* When the user clicks on various buttons on the menu, it should lead them to the correct part of the menu that they are looking for e.g. High scores, Info, Help etc. These will then need to be further developed into smaller sub modules so that they do the job they are supposed to do.

*General Update for the user interface –* A module of code is required to update the general details of the user interface. This includes highlighting the next wave of attackers and all the necessary details for the user.

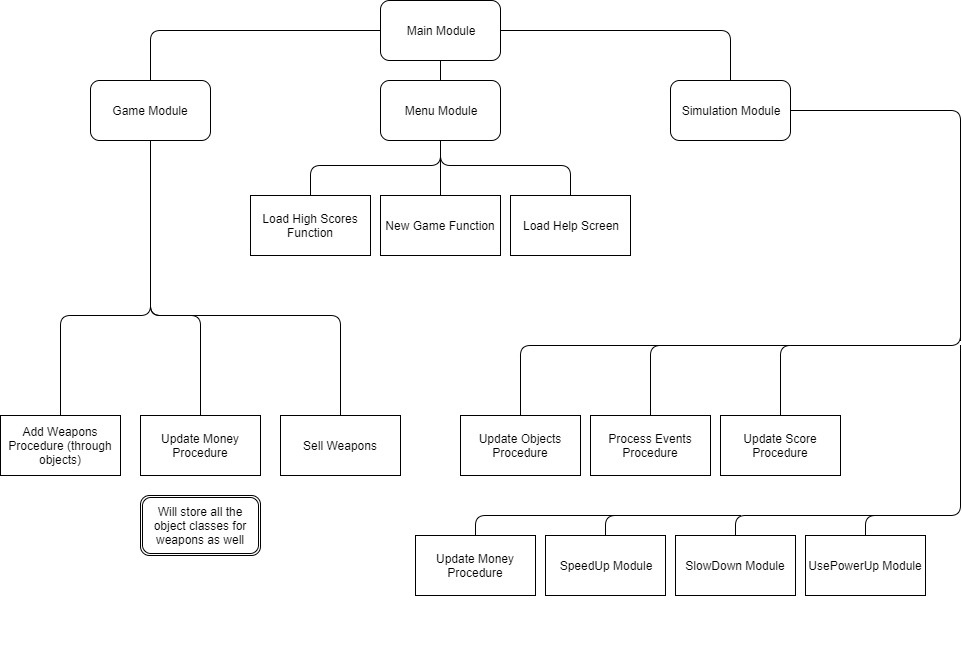
*Speed module –* A module of code should be able to increase the simulation speed and so will be closely linked to the simulation module.

*Select and drag module –* The user needs to be able to select the weapon they wish to purchase and then drag it to the right position on the grid that they wish to place it. This requires a module of its own to update the object’s values.

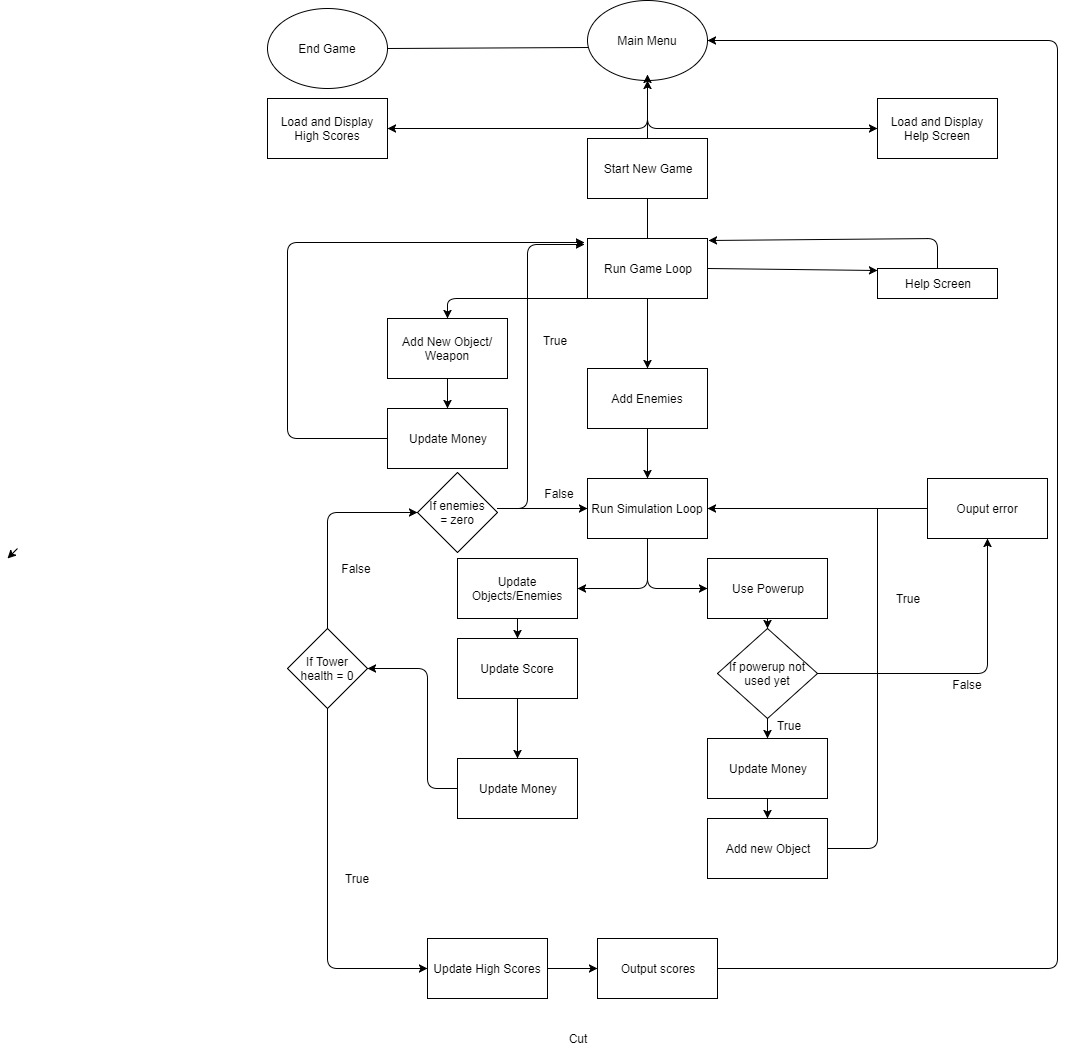
*Add/Delete new object –* Every time a weapon is purchased, a new object will need to be created using the correct class which requires its own module. Equally, should the user sell their weapon, this module needs to remove the object.

**Diagram to show the modules of code**

The relationships between these modules and what they will lead on to do are best shown in a diagram. This is the decomposition/stepwise refinement of the problem and how it will essentially follow through modules of code. By programming the most basic parts of the refined problems shown below, my programming task will be made far, far easier.



**The process of how the game will run through the form of a flowchart:**

To best way to understand the algorithms, classes and objects (as well as any other data structures) that I may need to implement, is to first understand how my game should flow and the steps my game will go through. This should help me formulate a plan of attack in terms of how the modules of code I came up with previously can be created. The flowchart follows a similar structure to the module relationship diagram shown on the previous page:

This structure is a good structure for a game:

When the user starts, they will need a Main Menu to be present so that they can navigate their way through the game and access other features if they need to (rather than diving straight into the game). This main menu needs to be kept simple so that users don’t look at it and get confused by it. Therefore, three options from the main menu should be fine.

The High Scores and Help Screen options can be accessed from the main menu and are standalone processes (as in they do not lead on to another process). This enables the user to simply load the information they want and once they are done, they are able to return to the main menu as when they click on these options, they only expect to see the information they want and return to where they started. The third option is to start a new game which will put them into the game loop. This game loop is necessary to keep the game running in time and on track.

The game loop has two potential loops. One is the help screen again. This is because the user may not have necessarily looked at the help screen when given the option to on the main menu and may still need help. To avoid them having to unnecessarily end the game and return to the main menu just for help, a help screen in the game loop will help them get help even during gameplay. The second potential loop is when they add a new weapon. This is because the entire point of the game is to place weapons to help your defence. This loop will add the new object in the position that the user drags the object to and will then update (deduct) money from their funds. It is important to deduct money to ensure that the user is not able to spend more money than they actually have.

Once the “play” button has been pressed, the Game Loop is broken out of, and then enemies have to be added. This is to ensure that the simulation can run and that there is something attacking the user (otherwise the game is pointless). The game will then enter the second loop – the simulation loop. Once again, this has two potential loops. One loop comes from if they decide to use a power up. If this happens, a logical decision will need to be made. If a power up has not yet been used, they will be allowed to buy one and so money will be deducted and the power up object will be added. However, if they have used a power up already, they will be given an error message in words to that effect. This will then loop back into the simulation loop.

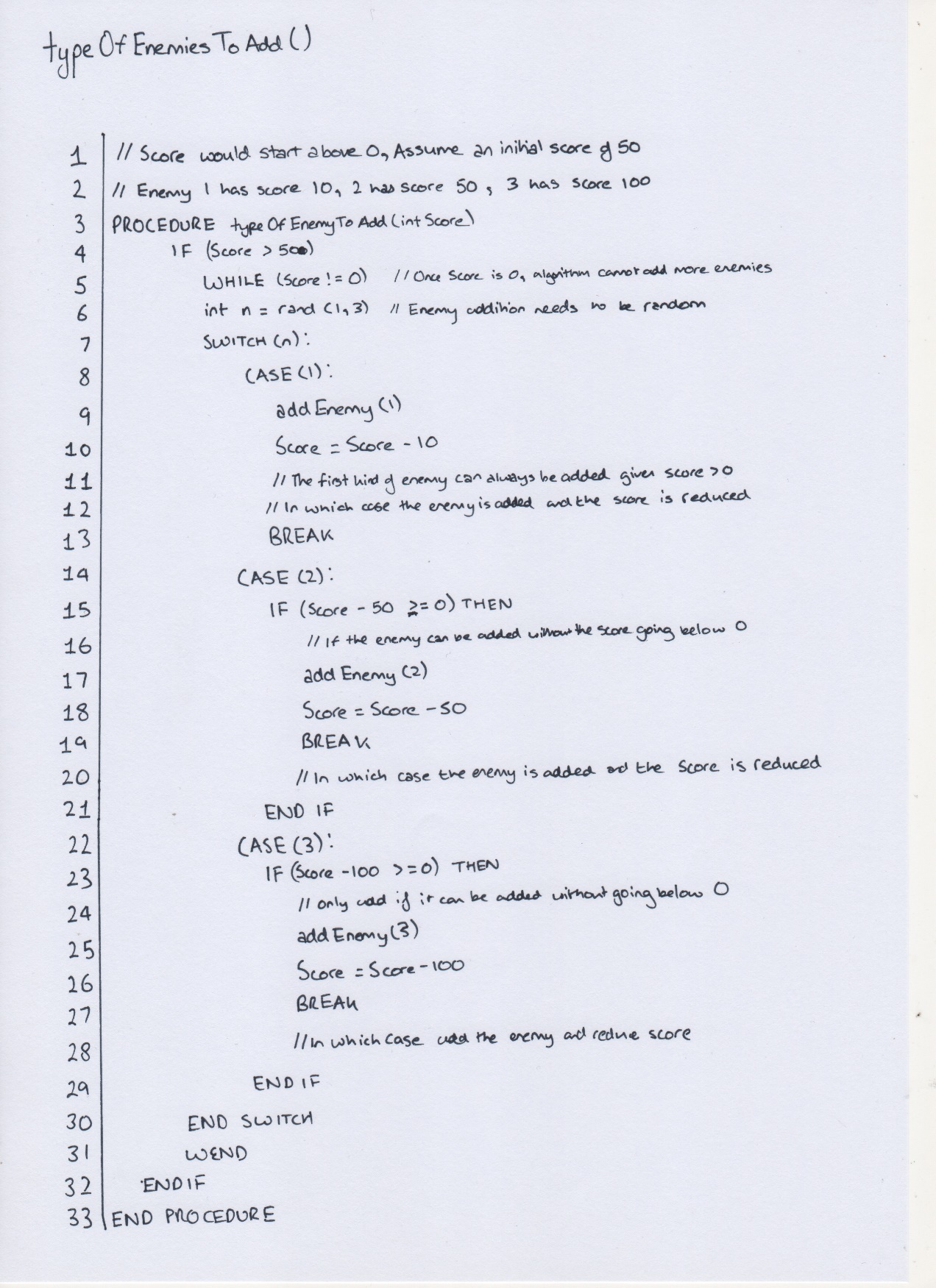
In all other events, all the objects on the map will need to move certain places, the weapons will attack the enemies if they are in a certain range and as such the enemy health (or the tower health) will decrease as a result. This will all be handled by the Update Object/Enemies module and is an important feature otherwise the game will not function properly. This will then need to be followed by an update in the score and money as for all damage to the enemy, the user will gain some score and earn money from it. From these updates, a logical decision then needs to be made again.

If the health of the tower is 0, then the user has lost the game. Therefore, if they made a new high score, the high scores will need to be updated and in all cases, the score they achieved will need to be output on the screen. The user should then be taken back to the main menu to decide what they want to do from there.

However, if the health of the tower is not 0, then the user has not yet lost. Therefore, another logical decision needs to be made in terms of whether or not all the enemies have died. If they haven’t, the simulation loop will just loop again, but if they have, then the user has passed this mission. The game needs to then therefore go back to the game loop to allow the user to add more weapons using the money they earned and hence improve their defence (and so making it a strategy game). Without these checks, the game may never end.

From the main menu, there is always an option available to end the game so that the user can quit and do something other than play the game.

**Pseudocode:**

**Enemy type algorithm** **-** One of the new additions I am making to my tower defence game is that every time someone plays the game, a random game is created. This is done by creating a random enemy line up every time the game is played, and so some pseudocode needed to be made before I could implement it (and hence it is part of the design stage)

**Enemy Type Algorithm – Dry Run**

Start with score - 250

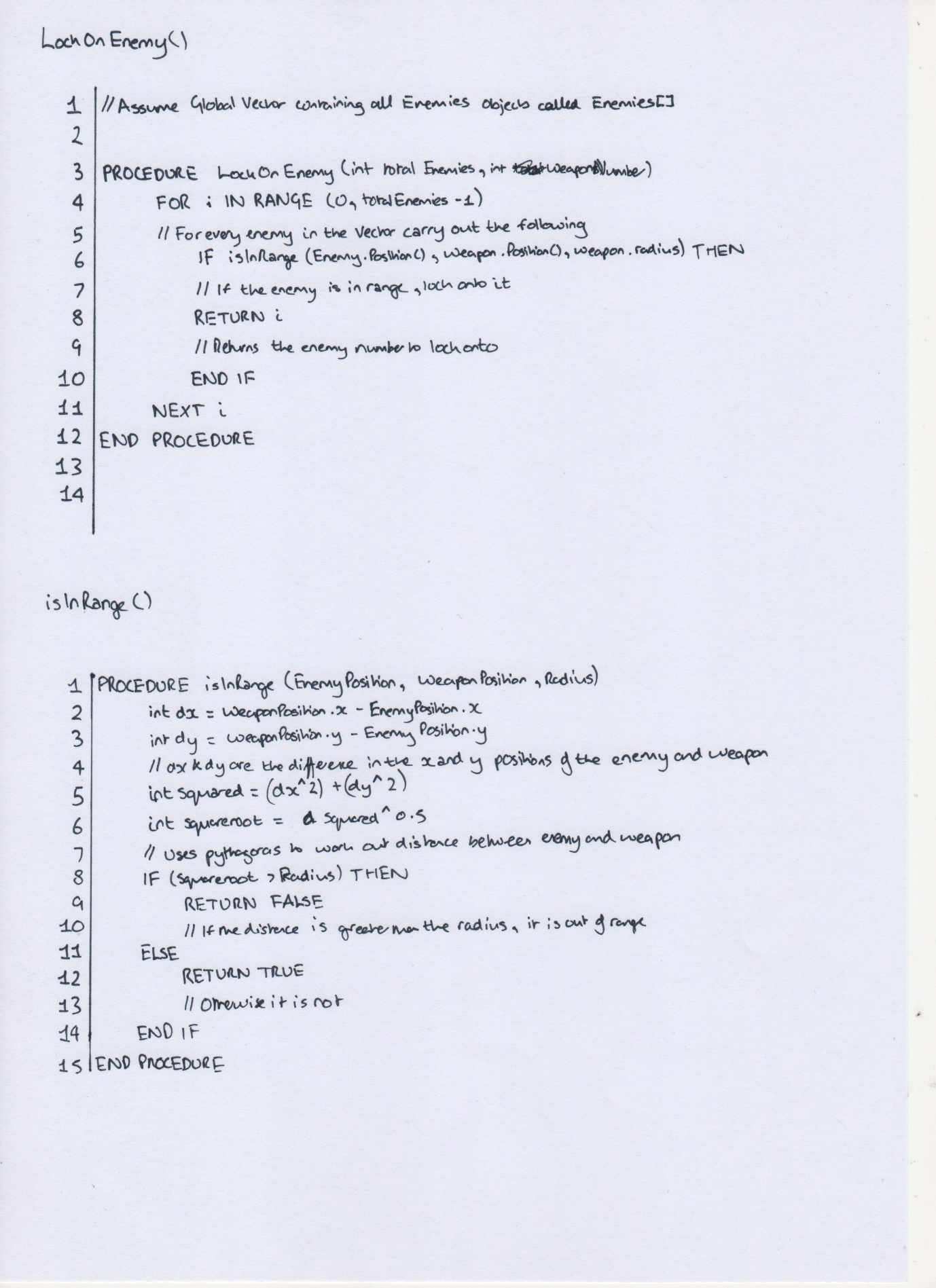
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Line Number** | **Score** | **n** | **addEnemy()** | **Score > 0** |
| 4 | 250 |  |  |  |
| 6 |  | 3 |  |  |
| 25 |  |  | 3 |  |
| 26 | 150 |  |  |  |
| 31 |  |  |  | TRUE |
| 6 |  | 1 |  |  |
| 9 |  |  | 1 |  |
| 10 | 140 |  |  |  |
| 31 |  |  |  | TRUE |
| 6 |  | 2 |  |  |
| 17 |  |  | 2 |  |
| 18 | 90 |  |  |  |
| 31 |  |  |  | TRUE |
| 6 |  | 1 |  |  |
| 9 |  |  | 1 |  |
| 10 | 80 |  |  |  |
| 31 |  |  |  | TRUE |
| 6 |  | 3 |  |  |
| 28 |  |  | Cannot Add |  |
| 31 |  |  |  | TRUE |
| 6 |  | 2 |  |  |
| 17 |  |  | 2 |  |
| 18 | 30 |  |  |  |
| 31 |  |  |  | TRUE |
| 6 |  | 2 |  |  |
| 17 |  |  | Cannot Add |  |
| 31 |  |  |  | TRUE |
| 6 |  | 1 |  |  |
| 9 |  |  | 1 |  |
| 10 | 20 |  |  |  |
| 31 |  |  |  | TRUE |
| 6 |  | 1 |  |  |
| 9 |  |  | 1 |  |
| 10 | 10 |  |  |  |
| 31 |  |  |  | TRUE |
| 6 |  | 1 |  |  |
| 9 |  |  | 1 |  |
| 10 | 0 |  |  |  |
| 31 |  |  |  | FALSE |

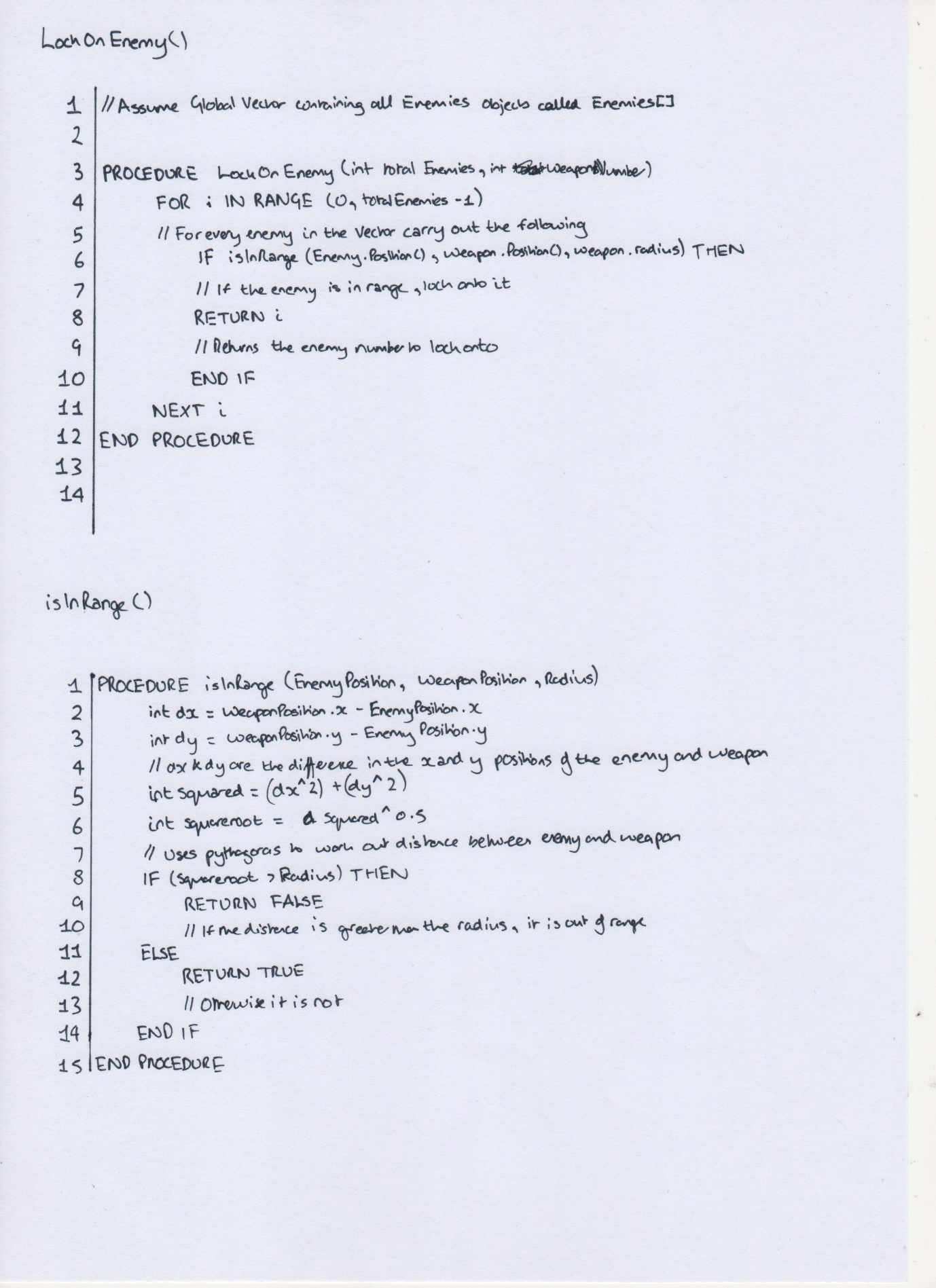
End of code. It added an Enemy 3, 1, 2, 1, 2, 1, 1 and 1.

Start with score – 3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Line Number** | **Score** | **n** | **addEnemy()** | **Score > 0** |
| 5 | 3 |  |  |  |
| 6 |  | 3 |  |  |
| 25 |  |  | Cannot Add |  |
| 31 |  |  |  | TRUE |
| 5 | 3 |  |  |  |
| 6 |  | 2 |  |  |
| 25 |  |  | Cannot Add |  |
| 31 |  |  |  | TRUE |
| 5 | 3 |  |  |  |
| 6 |  | 3 |  |  |
| 25 |  |  | Cannot Add |  |
| 31 |  |  |  | TRUE |
| 5 | 3 |  |  |  |
| 6 |  | 1 |  |  |
| 25 |  |  | Cannot Add |  |
| 31 |  |  |  | TRUE |
| 5 | 3 |  |  |  |
| 6 |  | 1 |  |  |
| 25 |  |  | Cannot Add |  |
| 31 |  |  |  | TRUE |

This case would cause an infinite loop. However, this should be fine as the game only deals with scores in multiples of 10 and so is an impossible scenario to occur unless the user uses a cheat engine in an attempt to cheat and edit their score.

**Weapon Lock Algorithm** – When a weapon fires, it keeps firing on a certain enemy until it ends up outside of its range. This means that there needs to be a function to lock on to a certain enemy until they either die or leave the weapon’s radius. Therefore, I created some pseudocode to find the enemy that needs to be locked onto.

This also led to the production of an algorithm to check if a weapon is in range which would be fed into by the weapon lock algorithm. This would use Pythagoras to compare the distance of the enemy from the weapon with the radius of the weapon, and if it was within range, it would only then tell the weapon lock algorithm to lock onto the weapon.

**Weapon Lock Algorithm – Dry Run**

**Case:**

|  |  |  |
| --- | --- | --- |
| **Enemy Number** | **x** | **y** |
| 0 | 140 | 600 |
| 1 | 60 | 200 |
| 2 | 30 | 40 |

Weapon 1 Position – (40, 60)

Weapon 2 Position – (80, 60)

Radius of 100 each

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Line No.** | **i** | **totalEnemies** | **weaponNumber** | **Weapons[weaponNumber] .radius > Enemies[i].position?** | **Return** |
| 3 |  | 3 | 1 |  |  |
| 4 | 0 |  |  |  |  |
| 6 |  |  |  | FALSE |  |
| 4 | 1 |  |  |  |  |
| 6 |  |  |  | FALSE |  |
| 4 | 2 |  |  |  |  |
| 6 |  |  |  | TRUE |  |
| 8 |  |  |  |  | 2 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 |  | 3 | 2 |  |  |
| 4 | 0 |  |  |  |  |
| 6 |  |  |  | FALSE |  |
| 4 | 1 |  |  |  |  |
| 6 |  |  |  | FALSE |  |
| 4 | 2 |  |  |  |  |
| 6 |  |  |  | TRUE |  |
| 8 |  |  |  |  | 2 |

isInRange() fed into by LockOnEnemy() algorithm

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Line** | **dx** | **dy** | **(dx^2 + dy^2)** | **Sqrt(dx^2 –dy^2)** | **Return** |
| 2 | 100 |  |  |  |  |
| 3 |  | 540 |  |  |  |
| 5 |  |  | 301600 |  |  |
| 6 |  |  |  | 549 |  |
| 9 |  |  |  |  | FALSE |
| 2 | 20 |  |  |  |  |
| 3 |  | 140 |  |  |  |
| 5 |  |  | 19600 |  |  |
| 6 |  |  |  | 140 |  |
| 9 |  |  |  |  | FALSE |
| 2 | -10 |  |  |  |  |
| 3 |  | 20 |  |  |  |
| 5 |  |  | 500 |  |  |
| 6 |  |  |  | 22 |  |
| 12 |  |  |  |  | TRUE |
| 2 | 60 |  |  |  |  |
| 3 |  | 540 |  |  |  |
| 5 |  |  | 295200 |  |  |
| 6 |  |  |  | 543 |  |
| 9 |  |  |  |  | FALSE |
| 2 | -20 |  |  |  |  |
| 3 |  | 140 |  |  |  |
| 5 |  |  | 20000 |  |  |
| 6 |  |  |  | 141 |  |
| 9 |  |  |  |  | FALSE |
| 2 | 50 |  |  |  |  |
| 3 |  | 20 |  |  |  |
| 5 |  |  | 2900 |  |  |
| 6 |  |  |  | 53 |  |
| 12 |  |  |  |  | TRUE |

This works as expected. The two weapons both choose to lock onto Enemy Number 3 and no other enemy as this is the only enemy within their range.

**Case:**

|  |  |  |
| --- | --- | --- |
| **Enemy Number** | **x** | **y** |
| 0 | 60 | 20 |
| 1 | 500 | 200 |
| 2 | 45 | 20 |
| 3 | 250 | 20 |

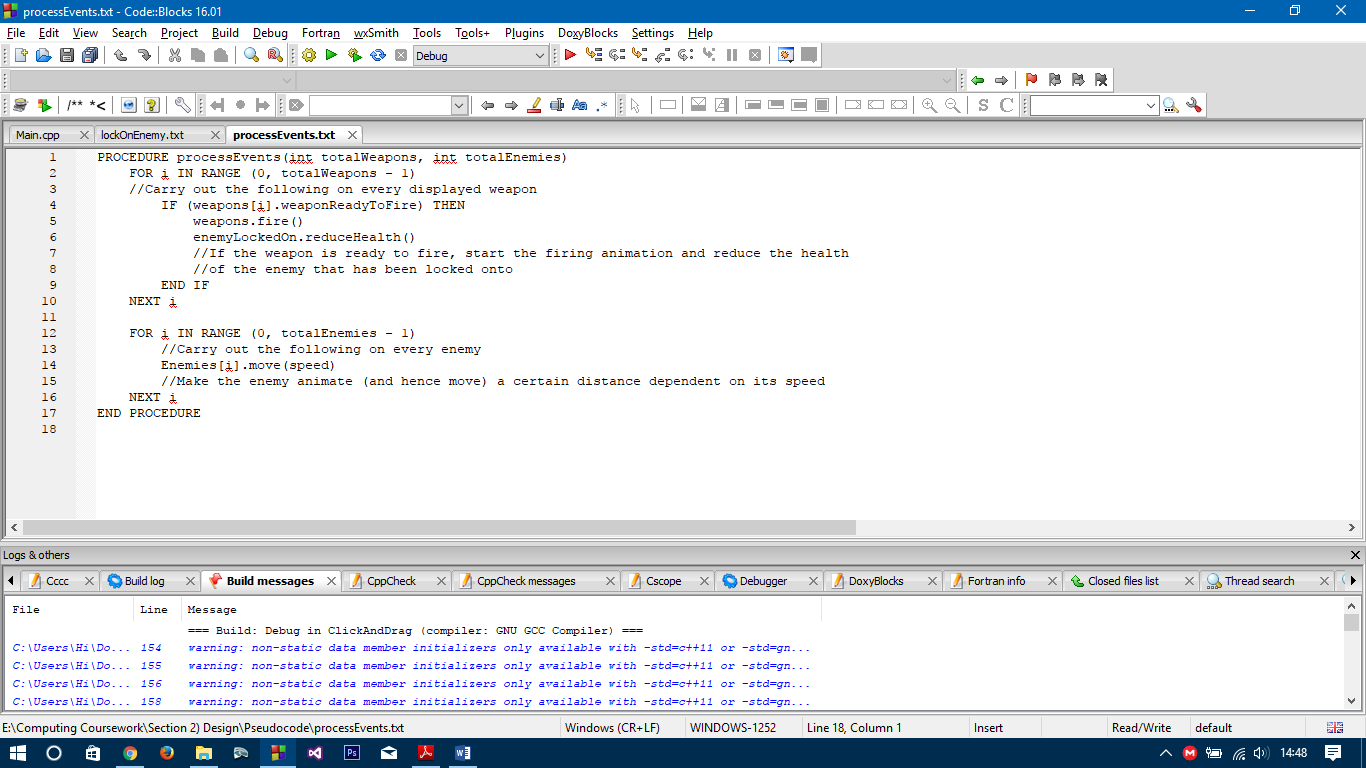
Weapon Position – (40, 20)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Line No.** | **i** | **totalEnemies** | **weaponNumber** | **Weapons[weaponNumber] .radius > Enemies[i].position?** | **Return** |
| 3 |  | 4 | 1 |  |  |
| 4 | 0 |  |  |  |  |
| 6 |  |  |  | TRUE |  |
| 8 |  |  |  |  | 0 |
| 4 | 1 |  |  |  |  |
| 6 |  |  |  | FALSE |  |
| 4 | 2 |  |  |  |  |
| 6 |  |  |  | TRUE |  |
| 8 |  |  |  |  | 5 |
| 4 | 3 |  |  |  |  |
| 6 |  |  |  | FALSE |  |

isInRange() fed into by LockOnEnemy() algorithm

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Line** | **dx** | **dy** | **(dx^2 + dy^2)** | **Sqrt(dx^2 –dy^2)** | **Return** |
| 2 | 20 |  |  |  |  |
| 3 |  | 0 |  |  |  |
| 5 |  |  | 400 |  |  |
| 6 |  |  |  | 20 |  |
| 12 |  |  |  |  | TRUE |
| 2 | 460 |  |  |  |  |
| 3 |  | 180 |  |  |  |
| 5 |  |  | 244000 |  |  |
| 6 |  |  |  | 493 |  |
| 9 |  |  |  |  | FALSE |
| 2 | 5 |  |  |  |  |
| 3 |  | 0 |  |  |  |
| 5 |  |  | 25 |  |  |
| 6 |  |  |  | 5 |  |
| 12 |  |  |  |  | TRUE |
| 2 | 210 |  |  |  |  |
| 3 |  | 0 |  |  |  |
| 5 |  |  | 44100 |  |  |
| 6 |  |  |  | 210 |  |
| 9 |  |  |  |  | FALSE |

This does not work as expected. This is because the algorithm finds two potential Enemies to lock onto, and instead locks onto the enemy that is furthest away. This can be fixed by putting in a break case on line 9 and so the weapon only locks onto the first enemy within its radius.

**Update enemy and other values loop** – The simulation loop runs throughout the running of the game and simulates/animates the movement of the weapons and enemies. Therefore, pseudocode regarding how the program will use polling to cause movement of the objects was required and so is shown below:

This was just a general loop which could not really be tested, and so I decided **not to dry run** this algorithm.

**Key Usability Features**

Not all users are experienced as one another – some users require more help than others in terms of how to play games, particularly if this is the first game they have ever played. Therefore, there need to be key usability features involved to ensure that the user knows what they are doing:

* Help screen available at all times – My help button will need to be available at all times, so when the game starts up and when the game is actually running. This help button will load up the help screen so that the user gets the help they require in terms of what the game is about and how to play the game. This will always be available in a visible part of the screen, and so the user will never be left alone without knowing what they should be doing.
* Error textboxes whenever the user makes a mistake – For example, if the user places a weapon on the actual path or outside of the given restrictions, then an error window will open to tell them the error they have made and (if necessary) what they need to do about it (along with a button to close the error screen once again).
* Speed buttons – There should be buttons to increase or decrease the animation speed when the game is running to ensure that the use doesn’t have to wait too long when they are happy with their defence (or vice versa). This ensures that the user enjoys the game.
* Clear design with big buttons – this is to ensure that everything that the user needs to play the game is clearly visible to them and they do not struggle to play the game. As a result, the game will not be frustrating for them. This will also be done through clear graphics.

**Variable, Data Structure and Class Listing:**

Some variables I know I will need are as follows:

* Enemy Speed – To ensure that the game runs at the same rate on all computers regardless of its specifications (particularly its clock speed), I will require an enemy speed. By keeping it a constant, it ensures that the objects move the same distance per second (regardless if it only travels this rate two seconds later). This is a integer as it stores a number.
* Total Weapons – A variable storing the value of the total number of weapons the user has added needs to be available. This is to ensure that the simulation loop is able to use polling to go through them all and hence simulate the firing of the weapon (if needed). This is an integer as you can only have whole weapons, not halves
* Total Enemies – A variable storing the value of the total number of enemies needs to be available as well so that each enemy can be worked with individually. These two variables are necessary as we cannot know before how many enemies or weapons before the game begins, and therefore a variable can overcome this problem. This is also an integer for similar reasons.
* Tower Health – As the game will only end once the tower’s health has reached zero, it is important to have a variable keeping track of the tower’s health so that once the game is required to end, the health of the tower can be checked and ended as a result. This is an integer as it is a number that is constantly being added to/subtracted from.
* Enemy Health – Similarly, the entire game runs on the basis that the enemies are attacked by the user’s weapons and once their health reaches zero, they are removed and the relevant score and money is added to the user’s variables. This is also an integer for the same reasons.
* Weapon Recharge Time – This is required as each weapon is different and so has different features such as recharge time: the time the user must wait before it can be used again. This is an object of the SFML Time class as it is a specific of the SFML Library
* Weapon Damage – Similarly, different weapons will do different damage to enemies, and so a variable storing the damage done by a certain weapon will need to be present. This is an integer.
* Money – A money variable is required to keep track of how much the user has available to spend on weapons and can be updated when weapons are bought or sold as well as when enemies are destroyed. This is an integer.
* Score – This is to keep track of how much score a user has earned so that there is an incentive to do better the next time they play. This is updated every time an enemy is killed. This is an integer.
* WeaponReturnMoney – When a weapon is sold, money will be returned to the user. However, each weapon will return a different amount when it is sold so a variable is required to store this value for the different weapons. This is an integer as well.
* EnemyScore – Each enemy will be scored/ranked according to how tough they are to defeat, and so a variable is required for each enemy (an integer)
* EnemyGainMoney – When an enemy dies, a variable needs to be accessed stating the amount of money the user earns as a result of destroying that particular enemy. (An integer)

Data structures I will need to use

* Vector to store multiple objects – Vectors are essentially dynamic arrays and are very useful for my project. This is because in theory, my game will need to hold an infinite amount of weapon and enemy objects rather than a predetermined number of weapons/enemies. Whenever you create an object, you need to give it a unique name, but vectors overcomes this problem as all you lost the unique name when you store it into a vector and instead you get an indexed value such as vectorName[1] will access the second object in the vector. In addition, vectors are dynamic so I can in theory hold an infinite amount of objects without worrying about having reserved positions in memory.
* Classes – The most important part of my game will be the classes I will use. The classes will define specific features of the game of different types of enemy and weapons, such as the damage they do, the health, reload times and more. Classes are key to this as the user will have multiple numbers of objects relating to their weapons and enemies and the easiest way to give a definition of an entity with multiple variables (without creating individual variables) is by using classes to define these unique variables.

Classes I will need to make

* Enemy (1/2/3) – For each of the enemies, (Enemy1, Enemy2 etc.) there needs to be a different class defining its features. One type of enemy will have a different score to another enemy. These enemy classes, however, will all inherently have the same variables within them:
  + Health variable to store their health so that when it reaches zero, the enemy is dead
  + MoneyGained variable to store the value of the money that will be added to the user’s total money once the enemy has been killed
  + Enemy Score variable relating to the score the player will earn when the enemy is killed

In addition, some methods for the enemy class that may be useful are:

* + updateHealth() to decrease the health when an object is hit by a weapon
  + getMoney() to keep the money variable private and have a get function to return the value of money
  + getScore() to return the value of the score

There will also have to be some external object destroyer function so that when the health is zero, the object is removed from the screen

* Weapon (1/2/3) – For each of the different kind of weapon (Weapon1, Weapon2 etc.), there also needs to be different classes defining its features. Once again, these classes should have similar variables regardless of which weapon they are:
  + Weapon Reset Time as more damaging weapons should not be able to fire as often as a less damaging weapon and so having a time variable should make this easier to implement
  + Reduce Health By variable to store the health value which when the weapon is fired, it will reduce an enemy’s health by. This is as you cannot know which specific weapon will attack which specific enemy before you play.
  + Return Money Variable so that when the weapon is sold, the user can get money back
  + Weapon Location to store the pixel location on the screen as to where the weapon has been placed

Some methods that may be required within the class are:

* + canFire() function will be required to ensure that the reset time has been achieved before the weapon fires again
  + getPower() to get the value which the enemy’s health should be reduced by (accesses the private Reduce Health By variable)

Like the Enemy class, this will require an external object setter and destroyer so that the display can be updated as required

* Power-up – The powerup is a standalone weapon which requires its own features in terms of variables and methods and therefore it requires its own class as well. The variables for this class will be:
  + isUsed will be a Boolean value ensuring that the powerup is only used once in a mission and not more
  + Reduce Health By variable like in the weapon class to check how much the health should be reduced by for enemies.
  + Location to store the location on the screen where the powerup has been placed

In addition, some methods will include:

* + canUse() to check if isUsed is true or false and if therefore the powerup can be used
  + getPower() to access the reduce health by variable

Class Diagrams of my classes:

**Weapon1**

**Enemy1**

getDamage()  
getPosition()  
setTexture()  
checkIfUsed()  
setPosition()

Is Used  
Weapon Damage  
Weapon Texture  
Weapon Position

getDamage()  
getCurrentTime()  
getReturnMoney()  
getBasePosition()  
getTurretPosition()  
setBasePosition()  
setTurretPosition()  
setBaseTexture()  
setTurretTexture()  
rotateTurret()  
resetTime()  
updateCurrentTime()

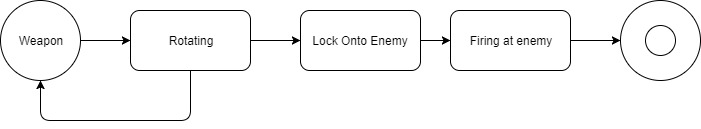
**Power-Up**

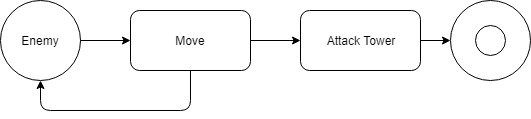
Weapon Reset Time  
Weapon Damage  
Return Money  
Base Position  
Turret Position  
Weapon Texture

getHealth()  
getMoneyGained()  
getScore()  
getPosition()  
setTexture()  
setPosition()  
updatePosition()  
updateHealth()

Health  
Money Gained  
Enemy Score  
Enemy Position  
Enemy Texture

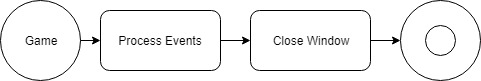
State Diagrams for the classes:

The Weapon State Class  
  
The weapon when in its running state will be rotating, locking onto enemies and firing at enemies.



The enemy on the other hand will only be focussed on moving and attacking the tower throughout its lifetime.

The entire game will also have a state diagram whereby it will be continuously processing events until the window is closed:



**Required Validation**

As part of the game, there will need to be validation to ensure that the user is carrying out correct procedures. Some of these need to be determined before I begin development:

* Checking if a weapon can be placed where it is being dragged. During the game, the weapon should not be placed too close to another weapon, on-top of another weapon or indeed on-top of the path. If this becomes a possibility, it may ruin the game. Therefore, it is necessary to have the ability to check if the weapon is being placed too close to either of these and if it is, then the correct feedback should be given e.g. an error message saying that this weapon cannot be placed here.
* Mouse and keyboard inputs will require validation in order to check which event to carry out. For example, I will need to be able to differentiate between a mouse click, mouse moved and the various keys on a keyboard. This will help with the save score screen when the user will need to enter their name as well as with the click and drag function for adding weapons to the map.
* Mouse input validation will also be required to check where the mouse is being clicked and if it is being clicked on top of a button (e.g. the pause button), the relevant function should run as a result.
* Checking if an enemy is dead or if the tower’s health is below zero. When an enemy’s health falls less than or equal to 0, then we will know that the enemy should no longer exist. Therefore, validating the enemy’s health value will ensure that the enemy is removed at the right time and the valid score and money is returned to the user as a result. In addition, if the tower’s health also needs to be validated, as if its health also falls below or is equal to zero, then we know the game is over and the game over events should start running.
* Comparing high scores. When adding a new high score to the high scores table, I will require validation to compare values already in the high scores database with the new high score value that needs to be put in. If the new score is higher, then its position will be higher (and so the position value be smaller) whereas the opposite is true for a smaller high score. This validation will ensure that the new value added into the high scores database has been put in the correct position.

**Specifics of the language and libraries that will be used to program this game**

For this game, I have decided that C++ is the most effective programming language to use. This is because C++ is very suitable for windows enabled devices which are the most common operating system on computers across the world. In addition, I know C++ better than other suitable languages such as Python, Java and other high-level languages. Finally, as C++ creates an executable file (in the file format .exe) and can take advantage of static linking (where a library is hardcoded into the executable itself), I will have the benefit of ending with a standalone application that can be run on many computers without the need for additional installations (as most of its requirements will be self-contained or pre-installed on most modern computers).

In terms of library code, I chose the SFML Library (Simple and Fast Multimedia Library) that is made specifically for C++ and gaming. This is because it contains classes and variables which work very efficiently and to a good standard. This means that my game will run very smoothly and quickly. In addition, it is very simple to use compared to other libraries and so it is the clear choice when it comes to game programming.

**SFML Specifics which will be beneficial for me:**

The sprite class within SFML has many key variables and attributes which will help me with my game development. For example, SpriteName.getPosition() (and the corresponding setPosition() function) is useful as it allows movement of the sprite throughout the game. In addition, it has other features such as setTexture that allow you to use your own images to add to the sprite without much hassle.

**Some User Interface mock ups and responses from my Stakeholders:**

I decided that now I was mostly done with my design, it was necessary to get the next most important part of the design stage – working out what kind of layout was most useful and well placed for my users. I therefore created some mock ups for the user interface screens on mockflow.com and posted these on my blog for my stakeholders to respond to.

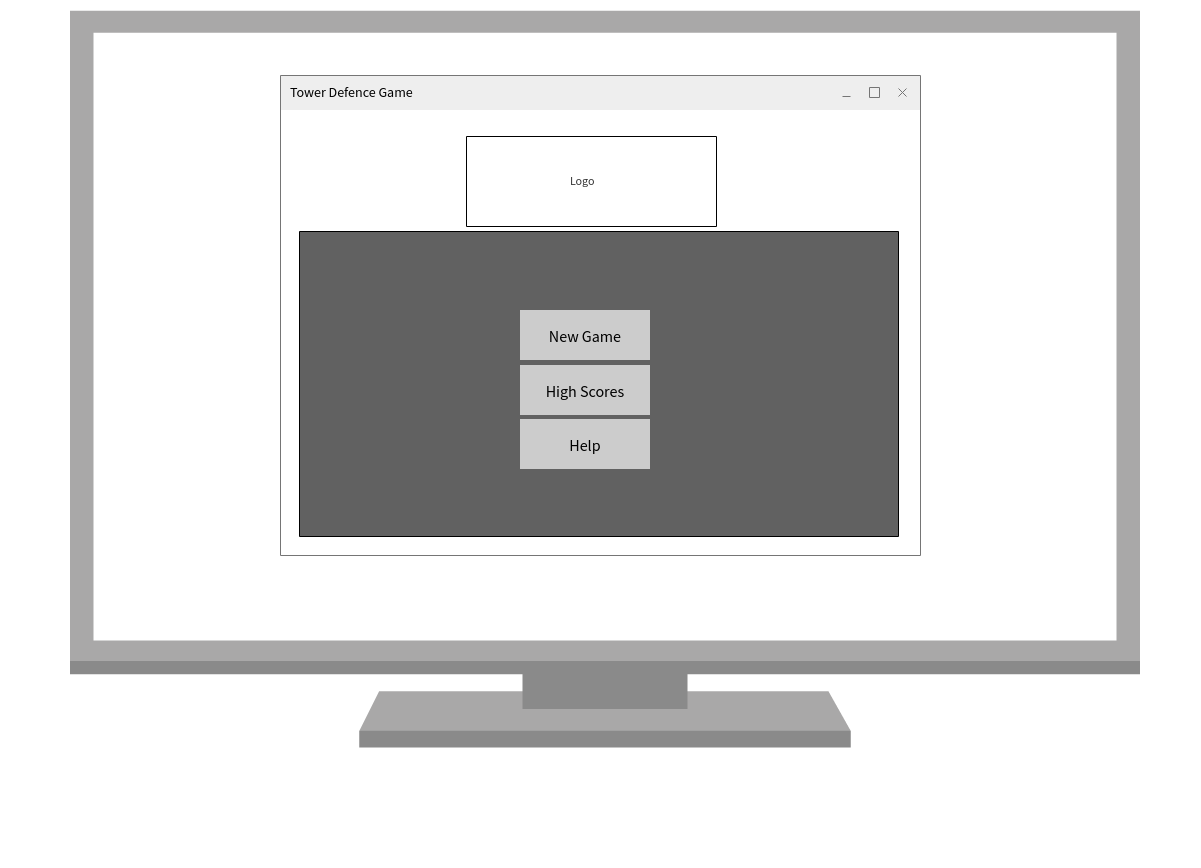
Based on my feedback, I decided on the following:

* The user layout is generally good and doesn’t need major improvement
* The actual gameplay area should probably be a bigger feature of the layout and not be controlled to such a small proportion
* On my main menu, I should fill up the empty spaces with some screenshots of the game itself to help with the visual
* A blue colour scheme will probably work best – it is a netural colour choice and most people’s favourite colour

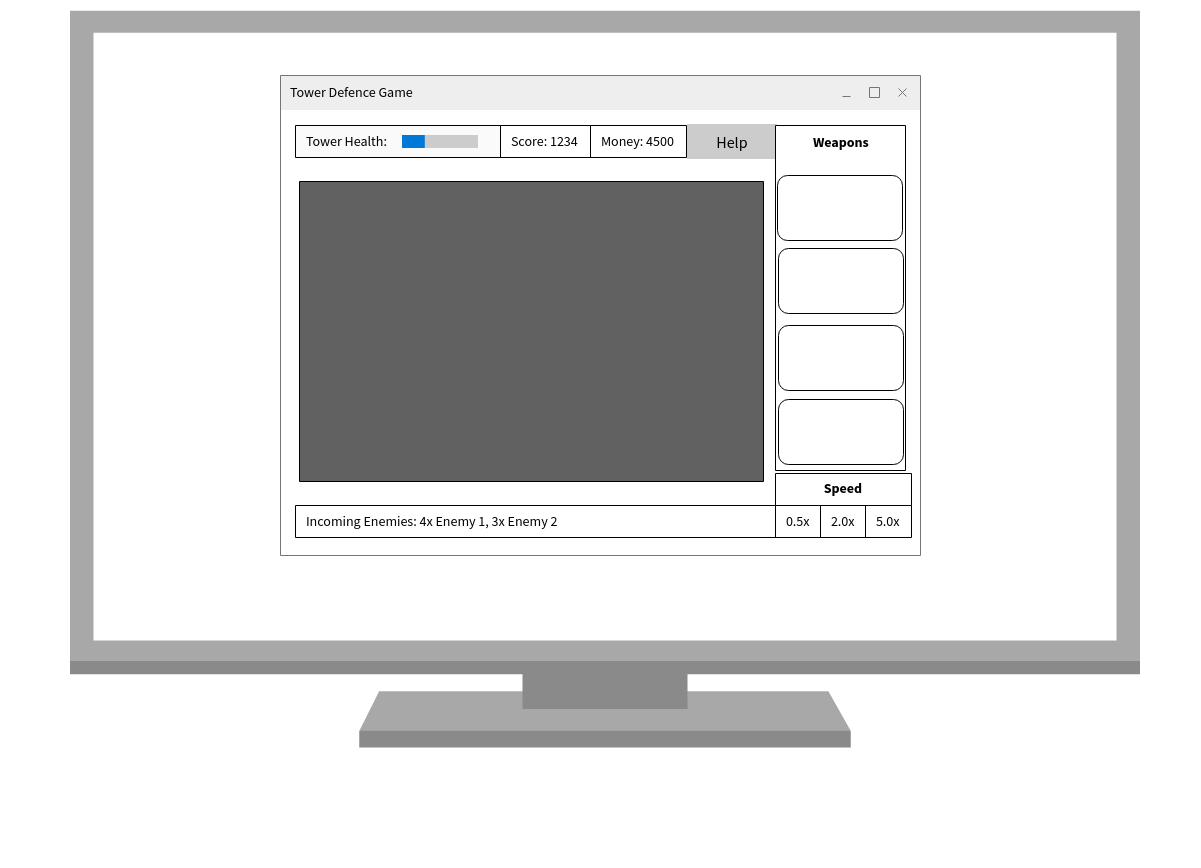
The mock ups and feedback are shown below

(<https://ghusharibcomputinggame.wordpress.com/2017/10/11/game-design/>):

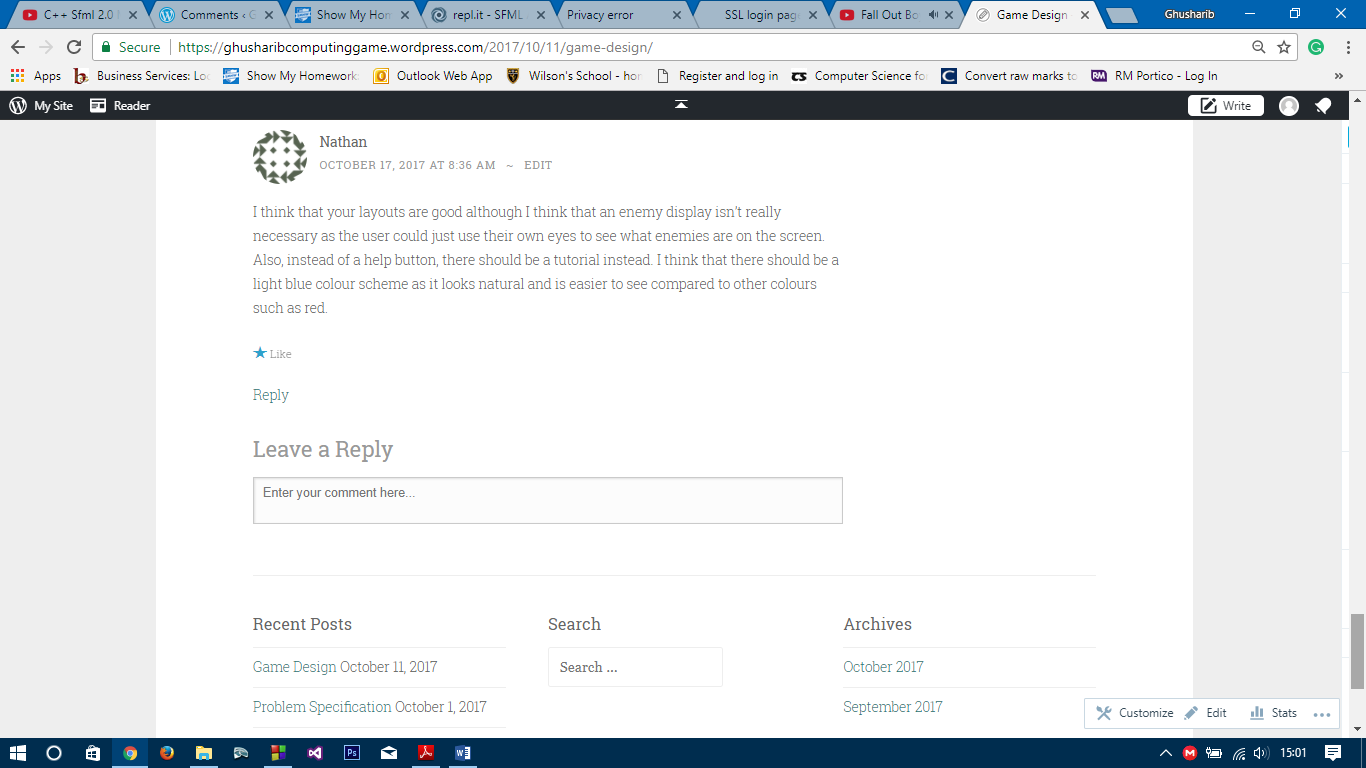
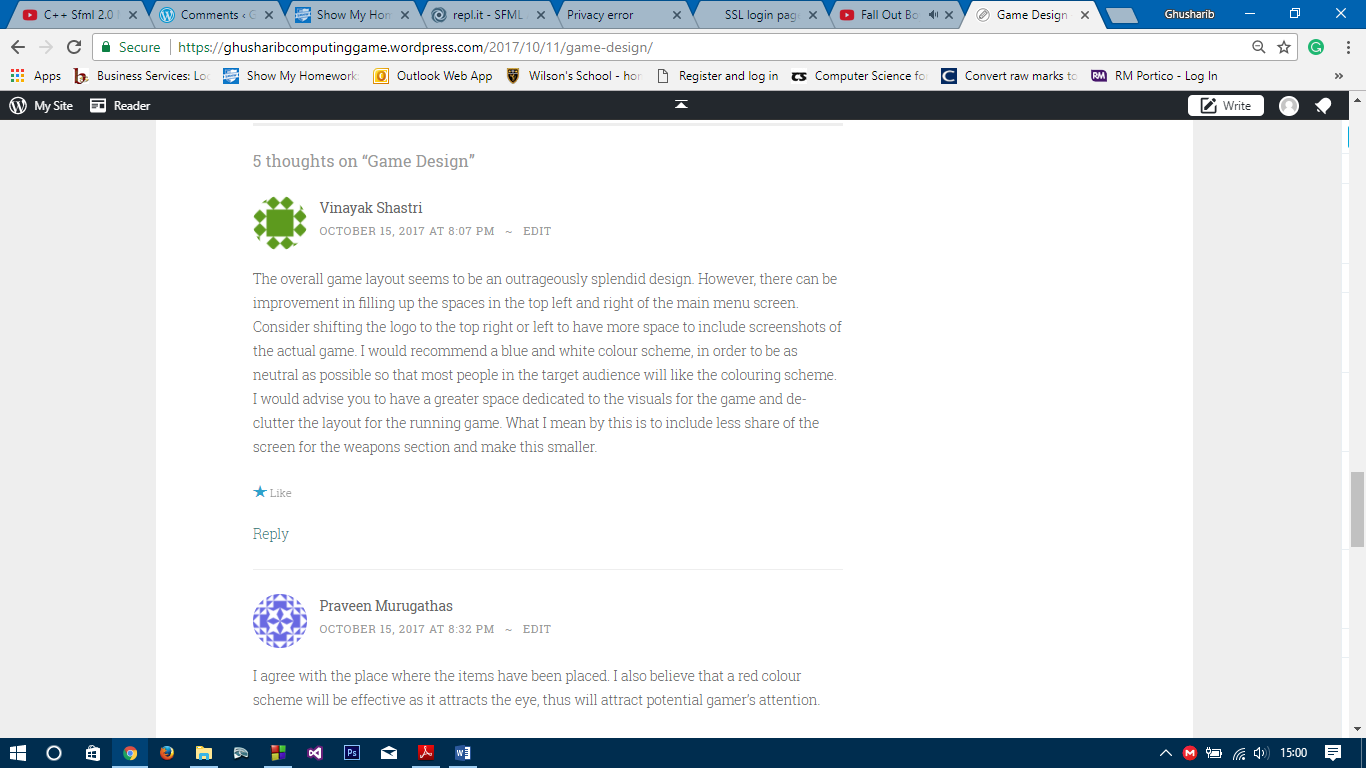
Main Menu Screen:



Game Running screens:



I also asked my user’s what kind of colour scheme they think would work well with this layout, and this was the response I got:



For the implementation, I have decided to follow 10 Iterative Stages (which are explained in more detail in my Iterative Development Section). The stages are as follows:

* Iteration 1 - Basic Window, Sprite, Images and Click and Drag with money
* Iteration 2 – Enemies and Enemy Animation
* Iteration 3 – Weapons Animations
* Iteration 4 – Collisions
* Iteration 5 – Power Up (and Corresponding Collisions)
* Iteration 6 – Game Loop
* Iteration 7 – Speed
* Iteration 8 – Menus, Buttons and Help Screen
* Iteration 9 – High Scores
* Iteration 10 – Aesthetics and Sounds

**See Appendix B for Test Plan During Development**

The first test data to be used is to be used during the iterative development process to test the functionality of the game so far. The reasons for these tests for each iteration are as follows:

For the first iteration, simple test data is required:

* The opening of the application needs to be tested to ensure that the game can actually start up, as if it starts up here, it will start up in later stages too
* The click and drag function needs to be tested here. This is because the first iteration is largely being used to add weapons to the map using user mouse input and so at this stage, this needs to be confirmed to be working before proceeding to the next stages (to avoid encountering problems later on)

Iteration 2 Introduces enemies and so the test data is related to enemy addition which should be automatic upon pressing “run”:

* The enemies that are added should be the right amount according to the player’s score using the enemies algorithm described previously. Therefore, it is important to make sure this is working at the end of this iteration
* The enemies need to be added one after the other to ensure that they don’t all bunch together in one place and therefore make it hard for weapons to fire at the enemy (and also, if enemies are all bunched together, the game will be much harder for the user)
* The enemies need to follow the shape of the map as that is what the user expects them to do. If they aren’t doing this, then the game will not work properly and so it is important this is tested now.

Iteration 3 again should also run automatically:

* The weapons need to lock on to the enemy using the lock on algorithm described previously. If this is not done, then the game won’t function
* The weapons also need to fire at the enemy given that it is within the correct distance
* The reset times are important as this is a major requirement of the problem specification and therefore needs to be tested at this stage

Iteration 4 involves adding collisions:

* When a weapon attacks an enemy, or an enemy attacks the main tower, it is important that the enemy or the tower’s health goes down, and so this needs to be ensured for all enemies and towers and checked to be true at the end of this iteration
* If health of a tower or an enemy reaches zero, there needs to be some sort of feedback and removal of an object and this needs to be ensured by the code.

Iteration 5 – Powerup

* The power up is a key feature of the problem specification to be tested
* The power up needs to ensure that it can only be used once per mission. This needs to be tested to ensure it follows these rules. Equally, it needs to be tested to make sure that it is only bought given the user has enough money to buy it.
* In addition, the corresponding collisions need to be tested as well as this is another weapon

Iteration 6 – Game Loop

* As the simulation needs to run automatically without user input, it needs to be ensured that it is doing so and be tested at this stage

Iteration 7 – Speed

* This is part of the problem specification that needs to be checked to ensure that when the user clicks on a certain speed button, it increases/decreases the speed to the corresponding speed value and doesn’t do something invalid

Iteration 8 – Menus, Help, Info

* All of these buttons and help screens need to be checked to ensure that they are working/functioning and are displaying the correct information. If they are not, then the user may not know what to do in a certain situation.

Iteration 9 – High Scores

* This is part of the problem specification and needs to be tested to ensure that the file is being accessed correctly and is displaying high scores in the correct order.
* It also needs to automate the high score display and add the high score to the correct place in the file by itself, and needs to be tested to ensure it does this

Iteration 10 – Aesthetics and sounds

* This is purely testing to ensure everything looks nice and sound effects that have been added play at the correct time. This is to ensure a seamless experience for the user

The testing during development is not limited to this, as this is only the black box test data and there will be a need for white box testing during development as well.

**See Appendix C for Test Data for use After Development**

This test data is largely testing against the problem specification. The purpose of this test data is for it to act as a form of acceptance testing, where each part of the problem specification is checked and if it has been fulfilled, it is ticked off.

Each of the test data refers to a certain aspect of the game. For example, the Menu section refers to all the parts of the problem specification that are to do with the menu. Each button on the menu needs to be tested to ensure it is doing what that button is supposed to do, such as “Start New Game” should start the game and allow the user to play, whereas the “Help” button should allow the user to gain access to any Help Information they require. This again is a form of black box testing.