|  |  |  |  |
| --- | --- | --- | --- |
| Module Code:  CS3S667 | Module Title:  Artificial Intelligence for Game Developers | | Module Team:  Mike Reddy |
| Assessment Title and Tasks:  (Re)Creating the Classics | | | Assessment No.  1 |
| Date Set:  30-Sep-2021 11:00 | | Submission Date:  05-Nov-2021 23:59 | Return Date:  03-Dec-2021 23:59 |

Student: Nathanail Kinnis ID: 30001979

Game selected: Caverns (aka Bubble Bobble)

**Task of the game**

Task of the game is to get as many score points as possible whilst protecting yourself from enemy bolts and robots by either dodging them or firing bubbles at them.

**Planning:**

Different milestones were set during the implementation, with the first milestone being a simple ai player running towards the pickups whilst also shooting bubbles. Second milestone was to add ways for the player to protect itself from dangers by either dodging, defensively shooting, changing direction, preventing jumps etc, to keep the ai alive as long as possible. Milestone 3 was to add more detail and explore most specific scenarios that the AI might encounter. Lastly, milestone 4 was to adapt the robot to not involve randomness, speed them up, allow them to shoot directly and when available towards the player, and make them extra challenging for the player ai to deal with. Most of the above went as planned, besides a minute milestone, which during that, a huge improvement for the AI player, which lead to achieving a high score of almost 900 000! Due to it being last minute, some of the comments and explanations may not fit (sorry for this). Unfortunately, due to running multiple applications simultaneously the machine operating the code crashed midway, and a significant amount of the last minute work was lost with some not been replicated again:(.

**Player**

The player can move left and right, jump, and shot bubbles. The bubbles in the initial design can have different distances to travel, which was based upon the duration the user held the space button down. The players can use the bubbles to trap enemies and block bolts fired by them. The player starts the game with 3 health points and 2 extra lives, with each bolt dealing 1 health point damage to the player. When player runs out of health points, they will consume a life and be respawned on top of the map and if they run out of no lives, the game ends.

**Enemy Robots**

The robot can move left and right and shoot bolts at random directions. There are two types of enemy robots in the game, the normal type which moves and shoots bolts at random times and directions, and the aggressive type which can defend themselves from bubbles the player fires at them (bolts from the normal robot can block bubbles, but they don’t shoot at them deliberately). Each level has a certain number of enemy robots spawned, with higher levels having more enemies and more chances of aggressive enemies being spawned. Once all enemy robots are killed, the game will allow the player to pick-up and fruits available in the level and then continue to the next level. Fruits will keep spawning randomly as long as there are enemy robots still alive

**Score**

Players gain score points through the fruits randomly spawned around the map or through the fruits dropped by the enemy robots bubbled. There 3 types of fruits in the game which reward different amounts of score points, with apple being worth 100 score points, raspberry worth 200 score points and lemon worth 300 score points. In addition, there are two more pickups available, the extra health which adds an extra health to the player and the extra life which adds an extra life. Both of these are only dropped from enemy robots only.

**Levels**

There are 3 level designs available which loop through each other once all are played. Levelling up doesn’t provide any benefits, only difference is the difficulty, as more enemies are spawned on higher levels, who are more likely to be aggressive, and will be stronger.

So why level up?

**Player AI**

AI Main Tactic: The player AI will bubble up all enemies but 1 in the first level and keep that enemy alive whilst farming with minimum danger, as there is only one robot to threaten it.

The AI will try to mimic the way a player would play the game, without gaining additional information that a normal player wouldn’t have access to.

This will be done using the Finite State Machine(FSM) technique. Though the A\* was also attempted to an extent, the FSM seemed a more appropriate and less complex solution to follow, as with A\* there were a lot of variables that were needed to be considered to create a correct path which would made the code too difficult to implement.

The main tactic above is achieved via two “mood” states , the killing state, which will be the initial state of the player, and the farming state. Whenever there is only one enemy on screen, the state will change to farming. As the AI doesn’t know the number of enemies being spawned, whenever there is 1 enemy on screen, the state will switch to farming, and if more enemies spawn, the state will switch back to killing.

In addition, the player AI will have an additional 3 Action states, the idle state, the moving state and the shooting state. The idle state is active whenever there are no fruits to collect and no danger around the AI, the moving state is active whenever the player is moving, and the shooting is active whenever the player shoots. The starting state is the idle state which can transit to the moving state only and be transited only by the moving state. The moving state can transit and be transited to both shooting and idle states. The shooting state can only transit to the moving state and can be transited only by the moving state.

Whilst on the moving state, the player AI will search and move towards the closest pick up to it. In case there is an extra health or extra life available, the player ai will move towards those regardless of how far they are. In the first map, there are two areas of the map which cannot be accessed, so the AI will ignore pickups spawned there.

In the case of the pickup being higher than the AI, looks if there is a block above to jump to, or when it gets closer to it on the X axis, checks if it can jump as high as it and jumps it if can. In the case of no block was found and it can’t jump that high, moves to a side whilst applying the same strategy above to reach the height wanted.

In the case of the pick being lower than the AI, the AI will move to a side until it reaches the wanted height.

During the killing stage, the AI will shoot bubbles based on the number of orbs available(with a maximum number of bubbles active to be 5), the fire rate between bubble shots and the amount of bubbles active compared to the number of enemies active, to avoid the ai spam shooting unnecessarily bubbles. The range of the bubble is based on the distance between the player and the danger shooting towards.

As the robots only shoot bolts on the platform they are on, the player AI must make sure to avoid being on the same platform as a robot.

The player AI is most vulnerable when changing platforms, either dropping down a platform or jumping up a platform, as it can’t do much while being mid-air. Based on this, every time the player attempts to jump to a fruit or is heading towards a drop from a platform, it will check for enemies/bolts on the platform its moving to and if there is any, prevents itself from moving to the platform.

Rule 0: Never be on the same height as a bolt or enemy

To achieve Rule 0:

Rule 1: If there is an enemy or bolt on the same height as the Player AI and close to the player AI, player jumps.

Rule 2: If there is an enemy or both on the height above, avoid jumping.  
Rule 3 : If there is an enemy or bolt on the height below the player AI, avoid dropping.

Rule 4: Always remember Rule 0

The approach prefers the safest way, with as a result sometimes delaying or not getting some fruit at all to avoid facing an enemy

There are special scenarios which can penetrate the rules. Such scenarios are, robots being spawned/dropped from bottom on top of the player, which in most cases results in the player getting damaged as the player ai has minimum time to react. Additionally, the player ai struggles to deal with multiple bolts coming its way, or bolts that are on many platforms, which restricts its choices to flee. These are scenarios which were attempted to be dealt with but due to inconsistency and overlapping with existing code, were left out of the final build.

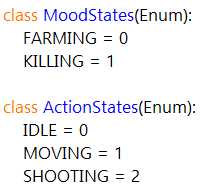
One of the most challenging parts the implementation of the player AI had, was how to deal with the robots when they were close to you. Quite a few ways to deal with this were implemented from simple ones such as simply jumping, to more complex such as search through all available escape ways and move towards them (current way of dealing with this)

**Player AI Implementation**

The states:

To put the states into code, the code uses enum classes for the mood and action states mentioned above, initializing each state with a number starting from 0. These will be used to set the state of the FSM.

MoodStates and ActionStates class enums code:



In addition, an additional class is created to encapsulate and hold all state related information for the player AI who will inherit this class and to be able to manipulate and utilize the states.

PlayerStates class code:



Each state has a different behaviour, thus for each state a new class is created. To do that, a parent class was created for both mood and action states, so each behaviour class inherits from them. Whilst this is not necessary, it give more clarity to the code.

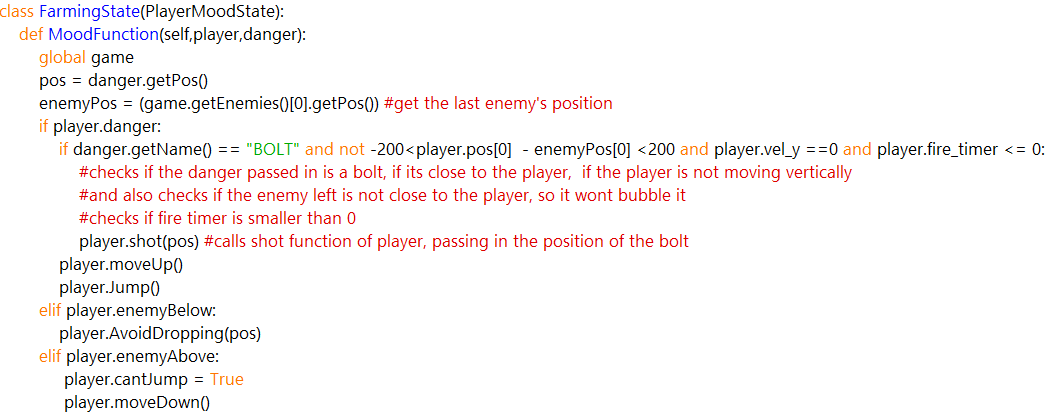
PlayerMoodState and PlayerActionState parent classes code:

Graphical user interface, text, application

Description automatically generated

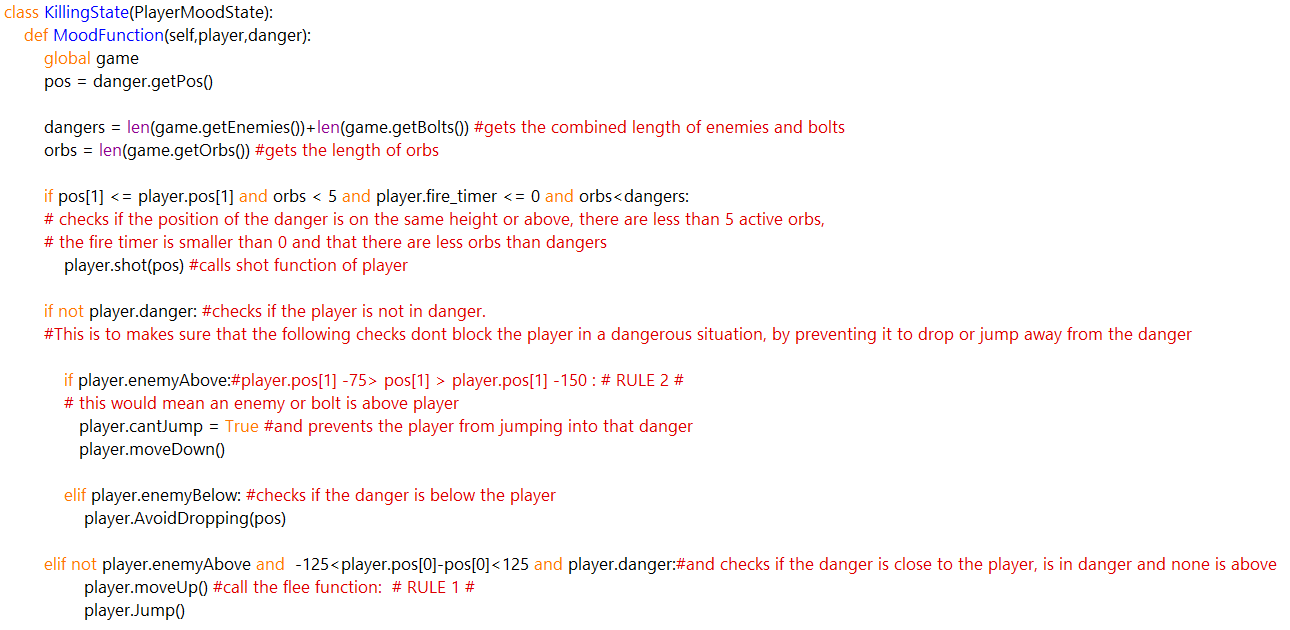
Each individual behaviour will modify the function inherited from the parent class and adapt it to perform its purpose.

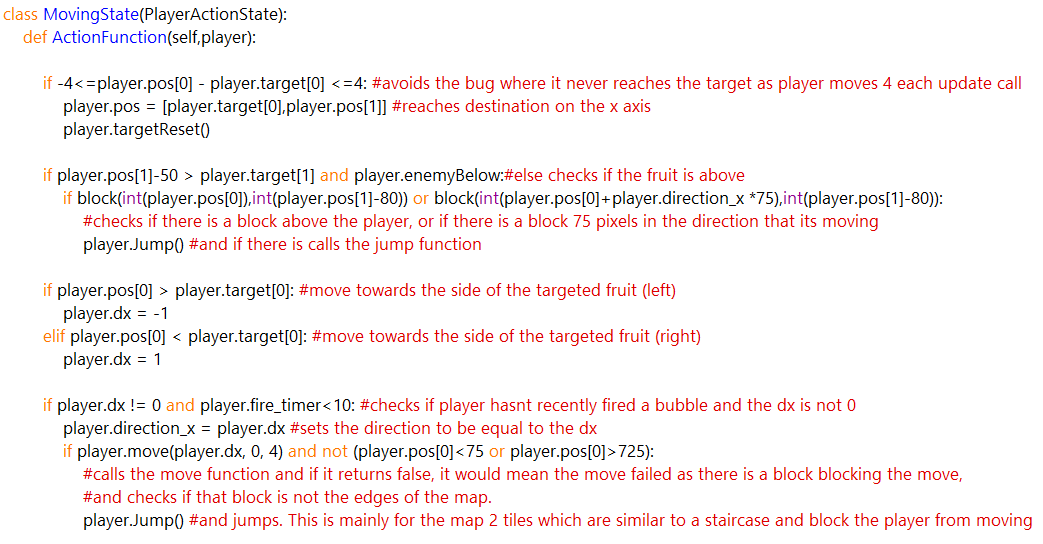
The farming state will either command the player to move down a platform, move up a platform, or stay on the same platform depending to the position of surrounding dangers. In addition, it will fire defensive bubbles towards bolts to block them

FarmingState class code:

The killing state will fire towards enemies on the same platform on above the player, and will evade dangers as above. (This was meant to be fused into one function but run out of time)

KillingState class code:



MovingState class code:

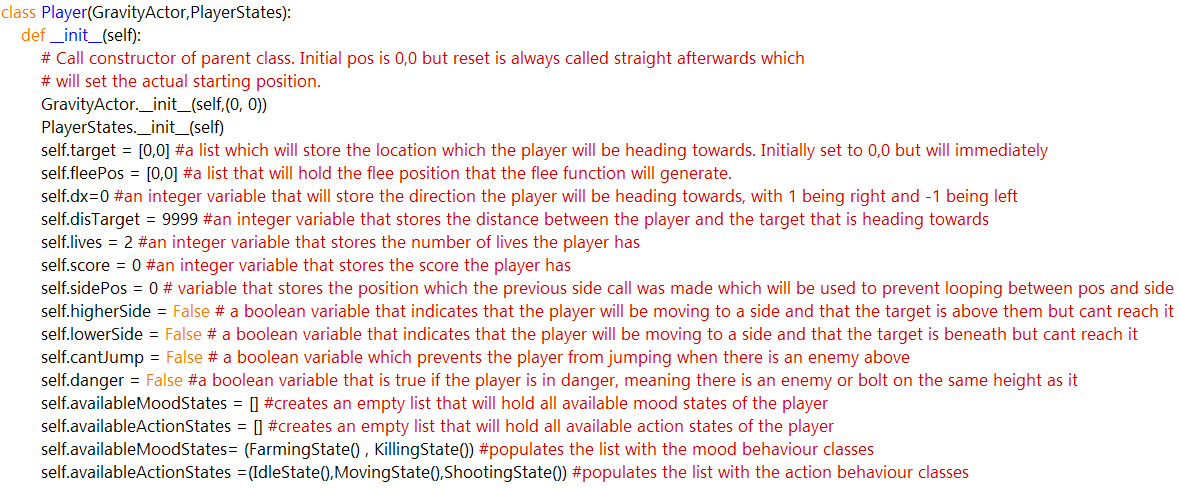
The idle and shooting behaviour functions are empty(with idle being empty on purpose), though the shooting function was attempted to be implemented but struggle to adapt the code to it and reverted the changes.

ShootingState and IdleState class code:

Graphical user interface, text

Description automatically generated

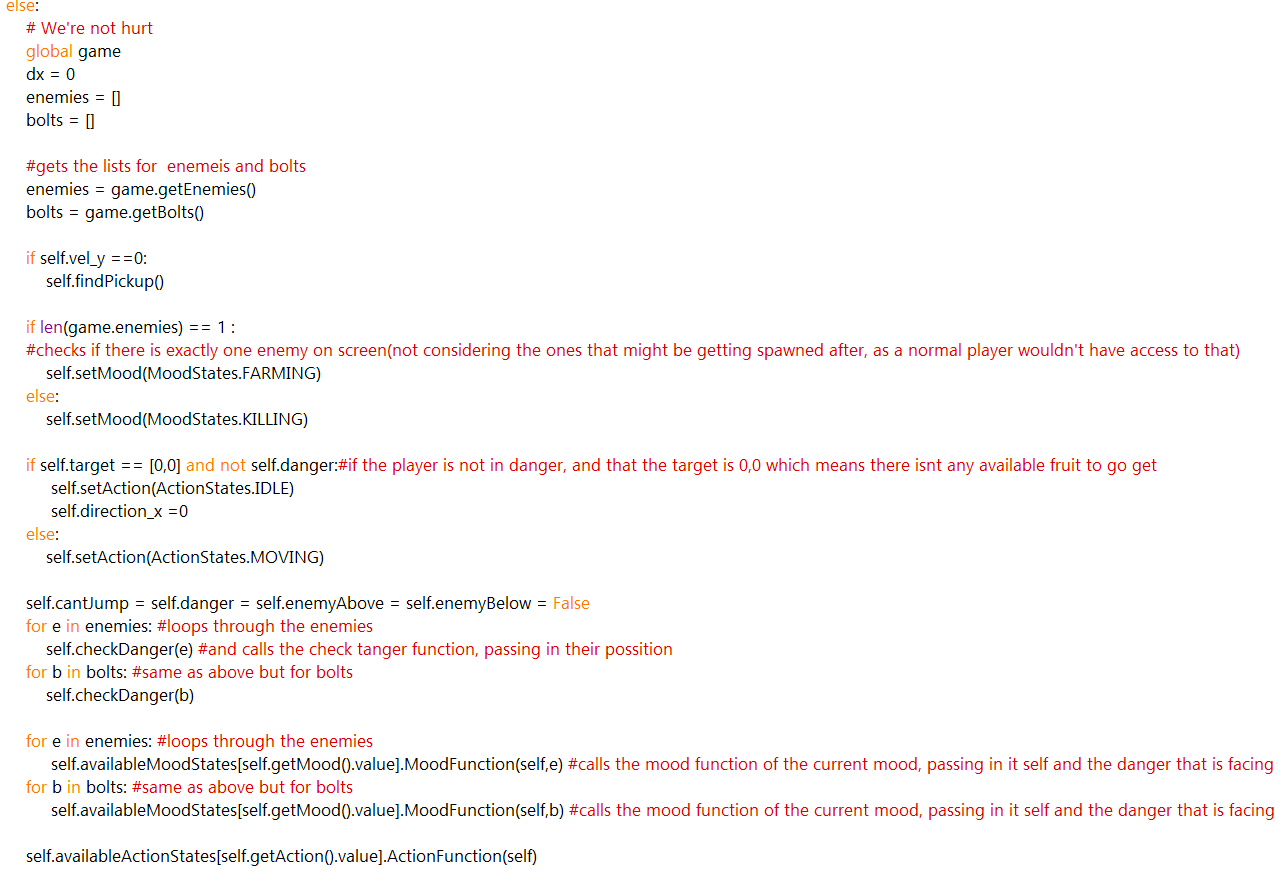
Player class variables:



Update Function

The update function handles state changes and all function calls

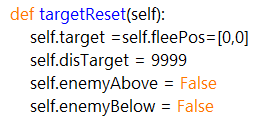
Update function code(changed):



General purpose functions:

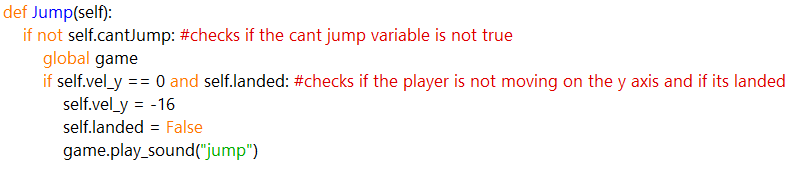
targetReset function: Resets the target and the variables associated with it

targetReset function code:



Jump function: Increases the velocity of the player on the y axis is conditions are met

Jump function code:



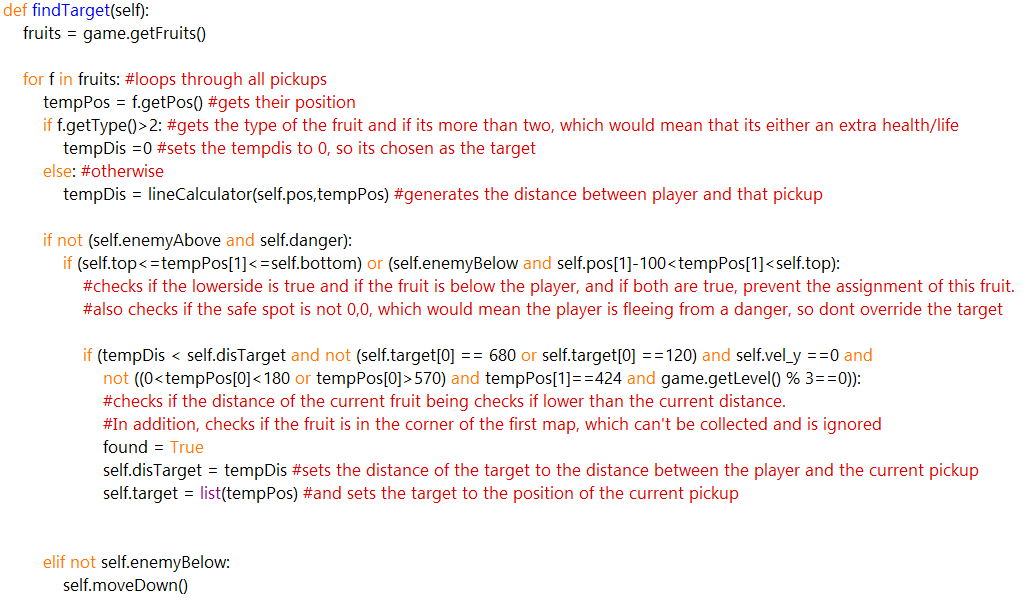
Helper functions:

To allow exchange of data between class objects, getters helper functions have been created for each class for specific data that simply return a data.

Pickup Collection:

The search for an available pickups is dealt by the findPickup function, which is called by the update function of the player class. It loops through all fruits, gets their type and position, and compares the distance between them and the player using the lineCalculator function and picks the pick-up which is closest to the player. In case a fruit has is an extra health/life, the distance to it will be set to 0 so the function always picks it as target for the player.

findTarget function code:

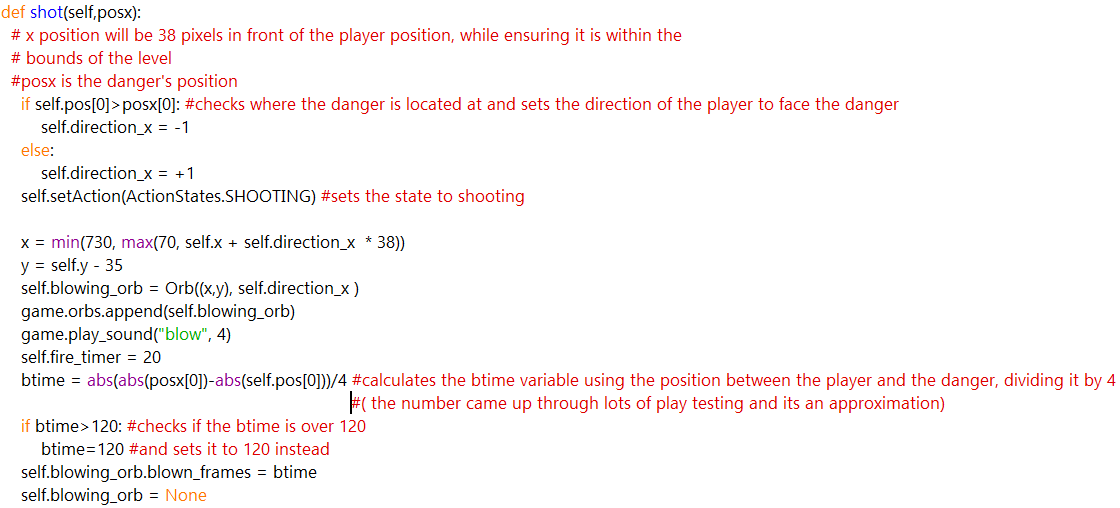


lineCalculator function code:



Bubbles:

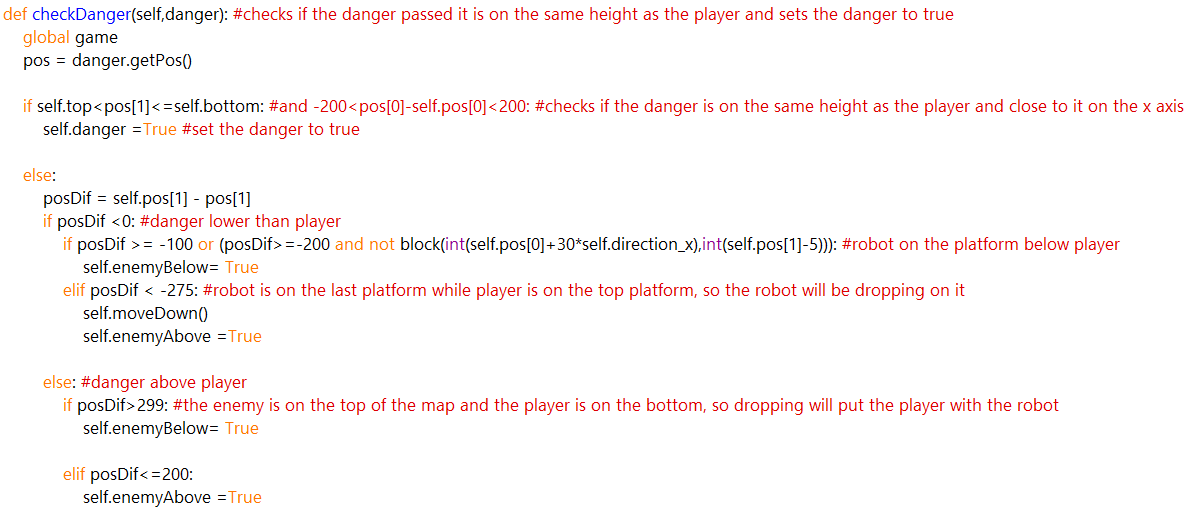
The player AI fires bubbles towards enemies and bolts that are on the same height or above as it. In the original game, the time a bubble would fly horizontally was based on the duration the player would press the space button, with a maximum value of 120. The player AI calculates an approximate time that the bubble would need to fly to reach its target when the bubble is shot. This is done in the shot function which receives the position of the danger that it needs to shoot at. The function is called through



Avoid Danger:

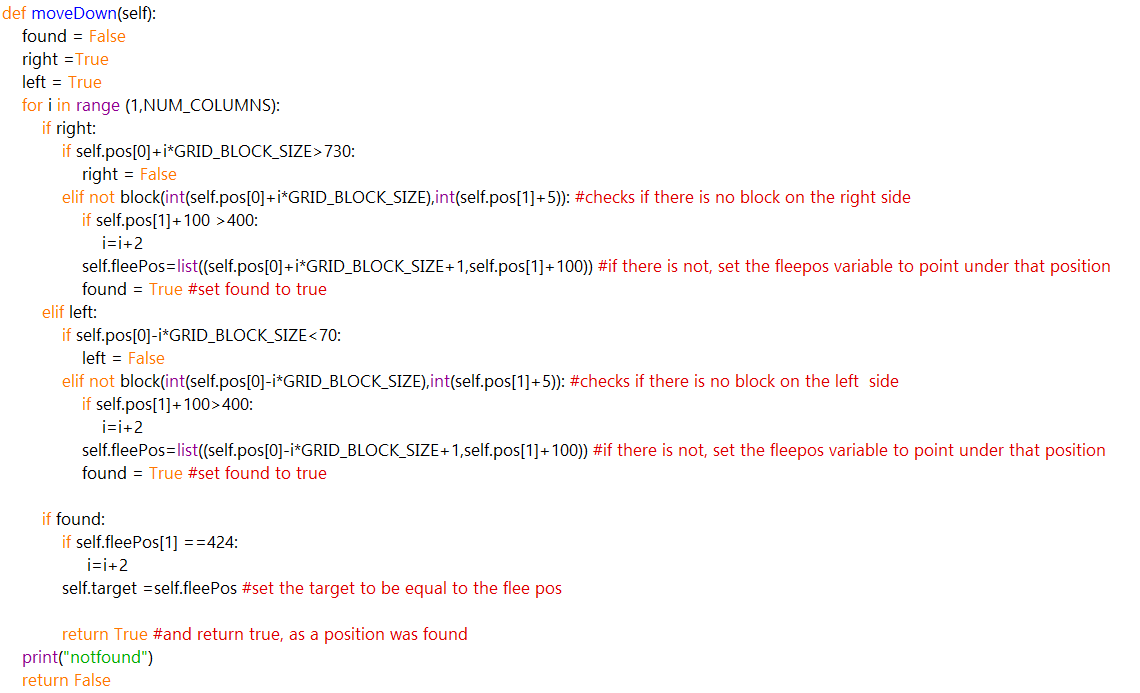
The AI’s main goal is to not get hurt and stay alive for as long as possible. To do so, the check function goes through all dangers and check whether there any that are on the platform above, below or the same platform as the player.

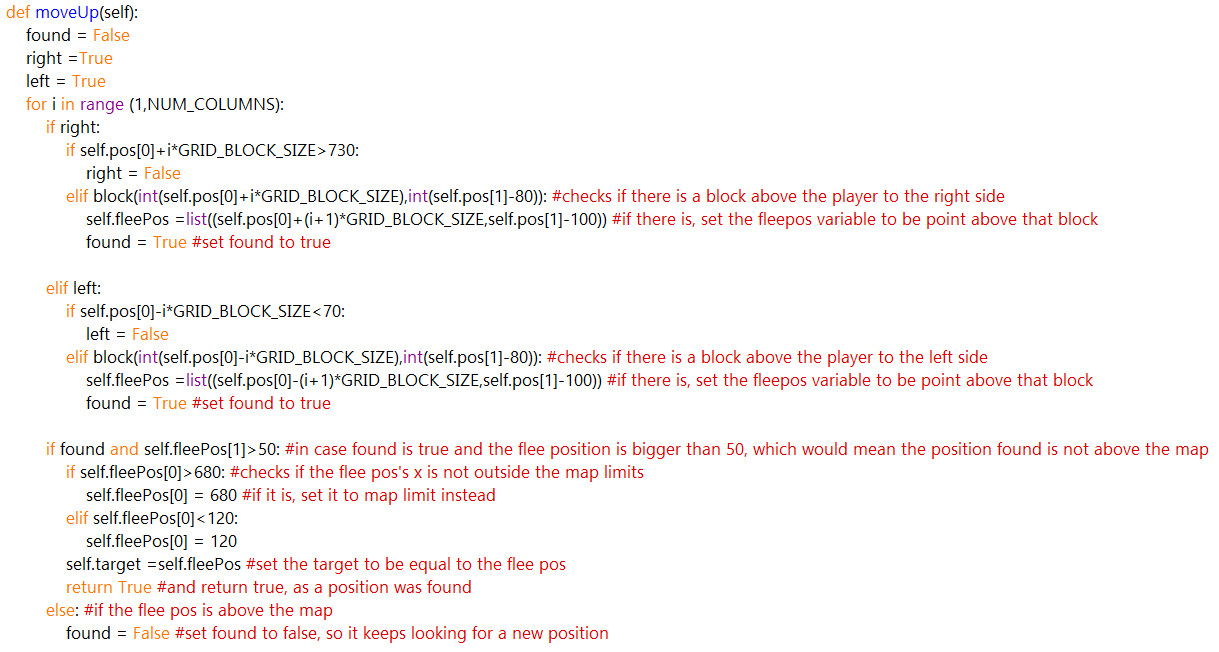
checkDanger function code:

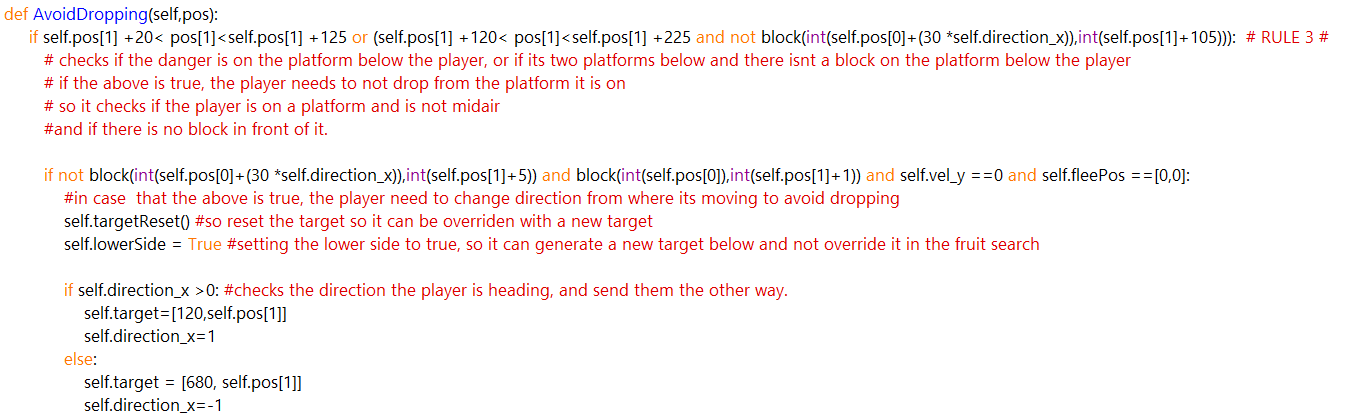
****

The move functions find the closest way to move either up or down a platform.

Move functions code:



****

AvoidDroppingFunction code

**Robot AI**

As for the robots, in the original game, most actions that the robots performed were random, including the direction of movement, direction of shooting and when to shot. To make the game more interesting and challenging towards the player AI, the robots will receive a no-random upgrade to its system, making it more powerful and precise. The new upgraded AI 2.0 will now be searching for the player on each platform they are on, and if found, will start shooting towards the direction of the player. In addition, to battle the farming tactics, the last robot will receive a speed boost so it can track and kill the enemy faster.

Goal of the robots:

Search for the player and once on the same platform as the player, move towards the player and shot bolts towards them.

To achieve this goal, the robots are added two different FSMs, one being the type of the robots which is either aggressive or normal type and it’s set upon the creation of the robot, and other is the action that the robot will perform, which can either be searching if the robot is not on the same platform as the player or found if it is.

Type FSM:

Action FSM:

The action states can swap between one another, whilst the type states are set once on the initialization of the robot and cannot swap transit after.

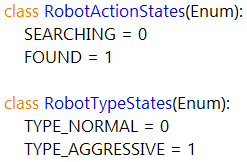
The action state is set to the searching state upon initialization, will search if the robot is on the correct platform, if its not it will move to a side of the map and try to find the player. Once the player is found (is on the same platform as the robot), the state switches to found and the robot will move towards the player, whilst firing bolts at them. If the player changes platform, the action state switches to searching.

**Robots Implementation:**

The states:

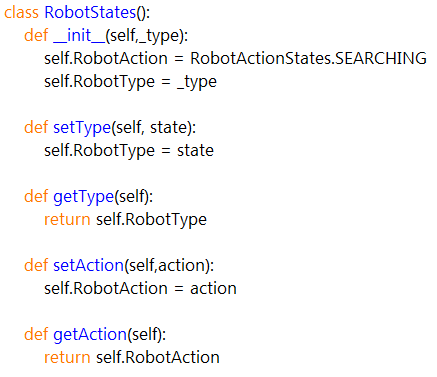
Similar to the player states, the code uses enum classes to represent the action and type states and will be used to set/get the state of the FSM.

RobotActionStates and RobotTypeStates enum classes code:



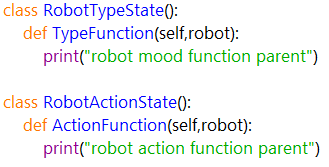
To use these, a RobotStates class is created and will hold the current state for each of the above, while also having helper getter and setter functions to obtain and set those states. The class will be inherited from the robot class itself.

RobotStates class code:



To make the individual behaviours for each state, create a parent class for each FSM that the state behaviours will inherit from. Again, this is not necessary but aids clarity of the code.

RobotTypeState and RobotActionState parent classes code:



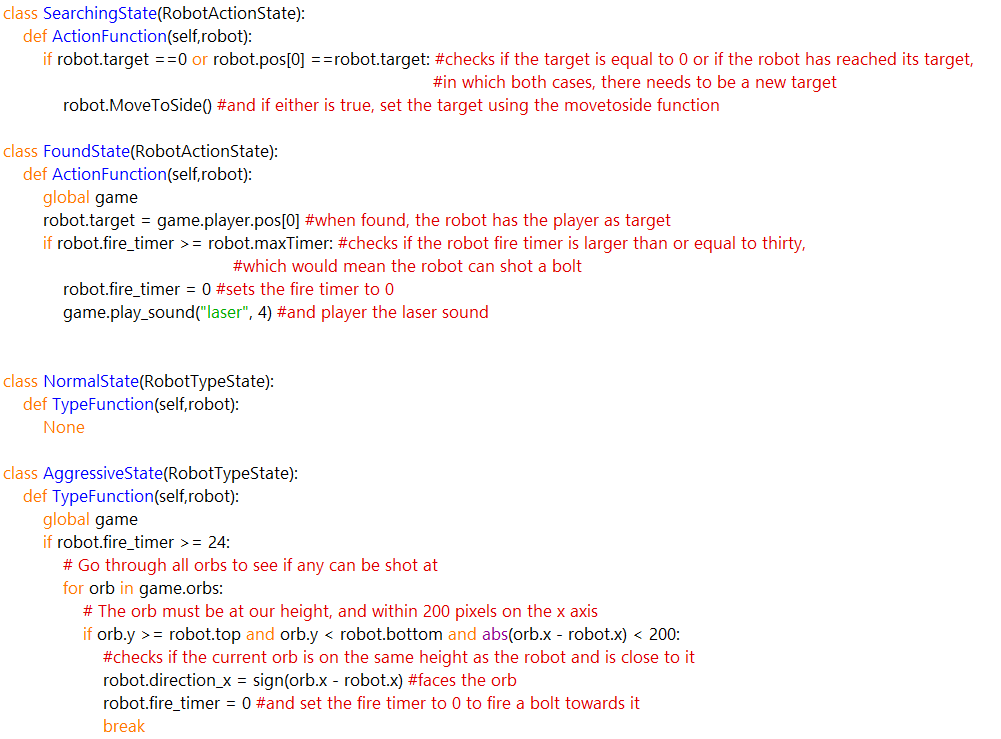
From the above classes, the behaviour classes will inherit and override the existing functions and modify them to their own behaviour.

The Searching state overrides the action function so if the target is equal to the position or is equal to 0, set the target by calling the MoveToSide function

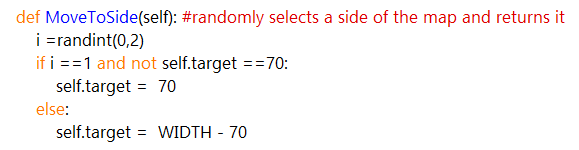
The Found state overrides the action function, and sets the target to follow the player’s x position, and also checks if the robot can shoot a bolt

The normal state overrides the type function but performs no specific task, as normal robots don’t have any unique features.

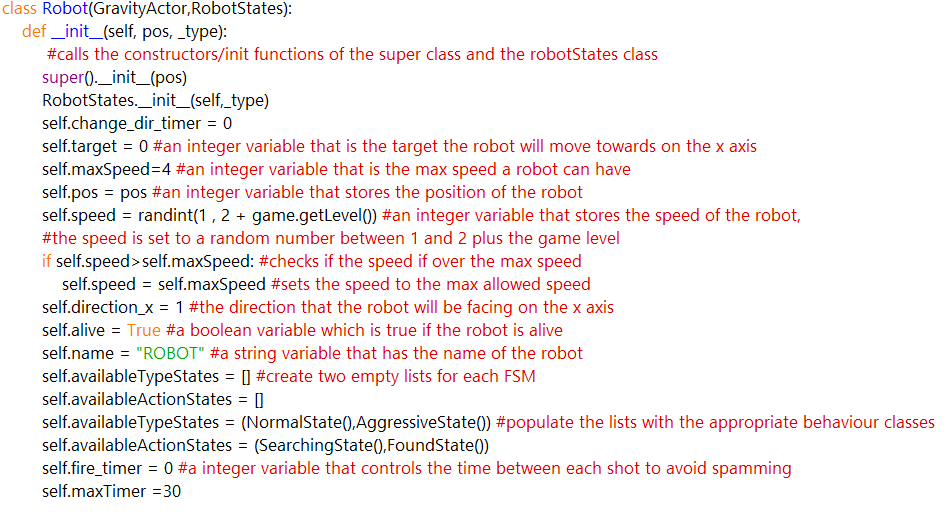
The aggressive state overrides the type function to allow the robots to protect themselves from incoming orbs by shooting bolts at them.



MoveToSide function code:

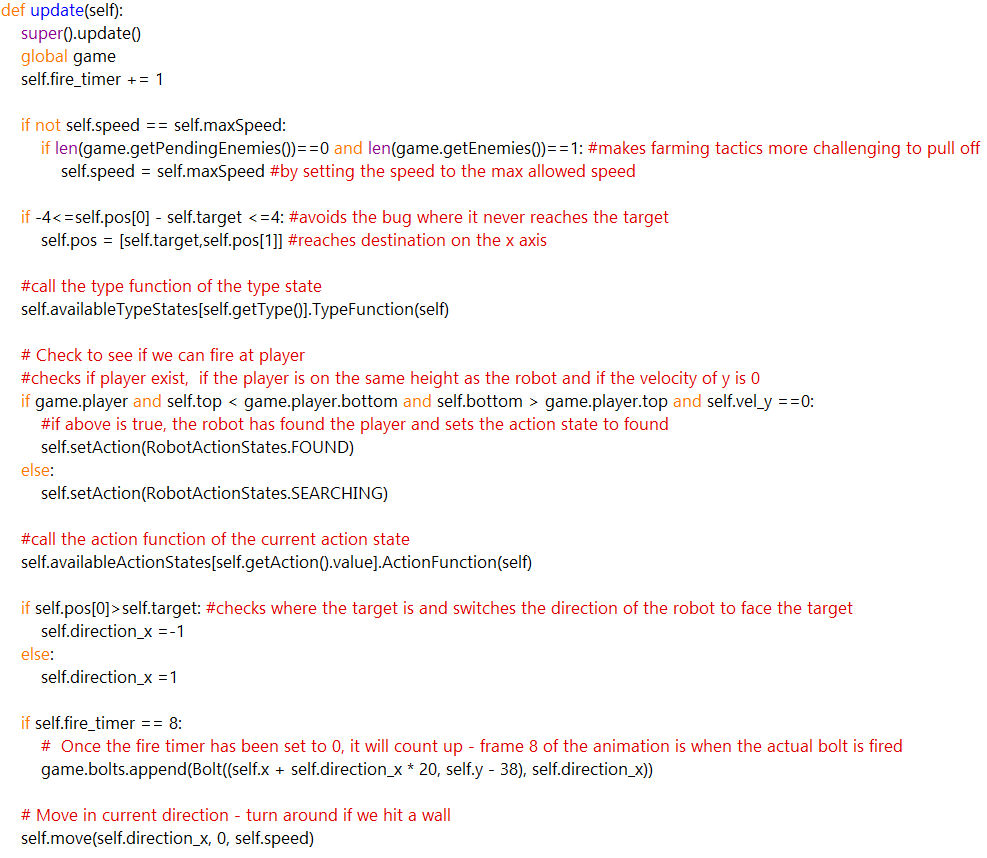


Robot class variables:



Update function:

The update function updates the states, variables, and performs all the call functions needed.



General Purpose functions

Graphical user interface, text

Description automatically generated

**Results/Testing:**

Through testing, a lot of issues were acknowledged and resolved. As the implementation continued, a result text file tracker was implemented to track the result for each run made. This was made during the early stages of milestone 4.

Result file writer code:

A screenshot of a computer

Description automatically generated with low confidence

During the earlier Milestones, a folder was created inside the project with screenshots of results of the testing processes.

The first milestone had an average of 100000 to 150000 score points.

The second milestone had an average of 300000 to 350000 score points.

The third-early forth milestone had an average of 50000 to 550000 score points.

The fourth and final milestone had an average of 15000 to 30000 score points.

The extra fifth milestone averaged 400000-600000 score points with a highscore of 896400

The player ai was getting away with bad moves unpunished, due to the randomness of the robots. Once the robots got upgraded, the player ai got punished for each bad move and the final score was dropped significantly.

In addition, an additional text file was created to trach the positions, of which the player took damage at, and through analysing the data, most of the damaged was dealt when the player was on the third platform (of the first level) and adapted the code accordingly.

Both text files can be found in the project folder in github, as well as the screenshot folder

The ai during the farming state deals with the environment efficiently and rarely gets damage dealt, but during killing state where there are more enemies to be dealt with, the ai player struggles and gets a lot of damage dealt during that stage. This was planned to be improved but eventually was left out due to time restrictions.