 **Faculty Of Computing and Engineering Sciences**

# Assessment Cover Sheet 2020-21

|  |  |  |
| --- | --- | --- |
| **Module Code:** | **Module Title:** | **Module Team:** |
| CS3S667 | Artificial Intelligence for Game Developers | Mike Reddy |
| **Assessment Title:** | | **Assessment No.:** |
| (Re)Creating the Classics | | 1 |
| **Date Set:** | **Submission Date:** | **Return Date:** |
| 28-Sep-2020 23:59 | 06-Nov-2020 23:59 | 04-Dec-2020 23:59 |

**IT IS YOUR RESPONSIBILITY TO KEEP RECORDS OF ALL WORK SUBMITTED.**

|  |
| --- |
| **Marking and Assessment** |
| This assignment will be marked out of **100%**.  This assignment contributes to **50%** of the total module marks. |
| **Learning Outcomes to be assessed** |
| As specified in the validated module descriptor [https://icis.southwales.ac.uk](https://icis.southwales.ac.uk/)   1. Understand the theory that underpins, and the pragmatic difficulties associated with, thedevelopment of a working AI game system 2. Evaluate the relative effectiveness of different approaches to AI for a given problem |
| *Awarded mark is only provisional: subject to change and / or confirmation by the Assessment Board.* |

The different types of algorithms for AI

There are a few types of AI algorithms that can be used, some have perfect standards while other include human errors into it, making them playable against actual human inputs, this is good as it will allow a game to be able to be beaten by players, making it fun. If there was a perfect AI that knew how to correctly do everything and never lose then that would be a problem with the game as it would never be able to be beaten by a human rending the game useless.

There are multiple different algorithms for AI but for this report I will only be choosing 2, this is because one of them can make human errors, and the other is perfect solution to AI so these two can contrast each other. With the way that they work.

FSM

What it is

FSM, or finite state machine, is a humanized type of algorithm that will put the controlled object into different states of actions. These different actions will have certain different commands or physical states to check for to move to a separate state, such as in the Bunner game that I have chosen for this project there are 7 different states that the bunny can get into, these are idle, move forward, move left, move right, move down, splat and splash, the last two are states that an AI are trying to avoid getting to. For an example of this, for a top down game like the older legend of Zelda games, if the player were to press “up” then the player would go up. Finite state machines are not only used for AI purposes and can be included into many different things.

Advantages

There are many advantages that finite state machines include, such as being able to make human errors and letting other players actually win, while also if advanced enough then can become a struggle for the player, this is good, as you can have many variations of difficulty on just this one algorithm alone (depending on how advanced the finite state machine is).

This algorithm is usually really simple to code and can be implemented into any program using any language.

Disadvantages

There are some drawback for this though, one of the main ones for this is that as it is state based depending on the state that the object is currently in, it limits the states it can go to depending on how complex the machine is.

A\*

What it is

A\* algorithm is a type of algorithm that tries to find the best route to get to a given destination, this is a “perfect” route, as it will calculate all other routes to get to the destination and then use the route that is best for the program that is being used for. This algorithm is mainly used for pathing in games. For the Bunner game that I was making this for, this wouldn’t have worked that well as our AI needed to have humanized logistics, instead of being able to run infinitely.

Advantages

The advantages of this is that it will check all of the given routes to a destination, and then choose the best one for the route to take. This is good for certain non-player based tools, where there is no input and you are trying to get the best place.

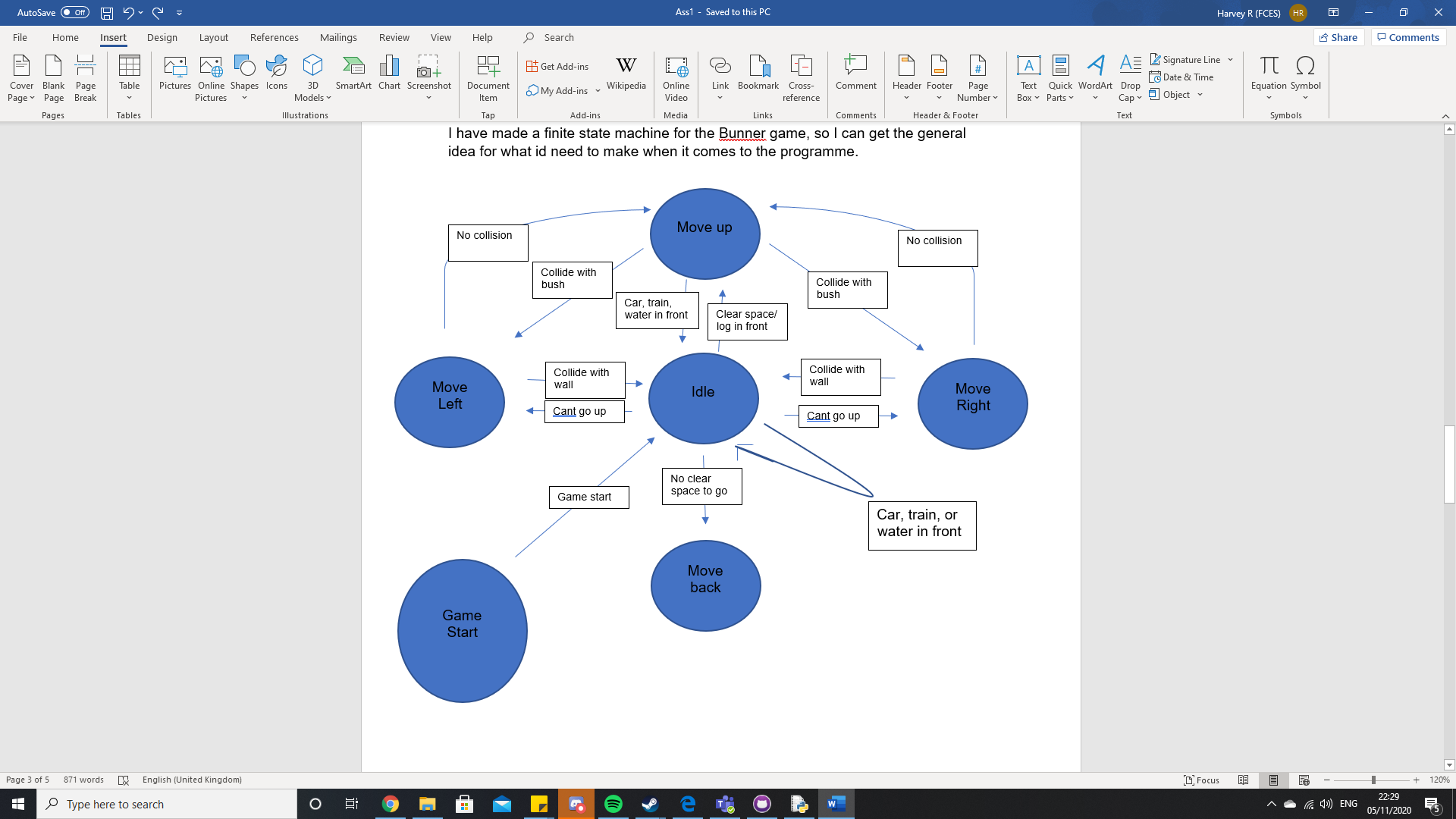
Disadvantaged

Can not become humanized, there is nothing in the code that allows a mistake in this algorithm as its all saved into the programme and it just chooses the best one.

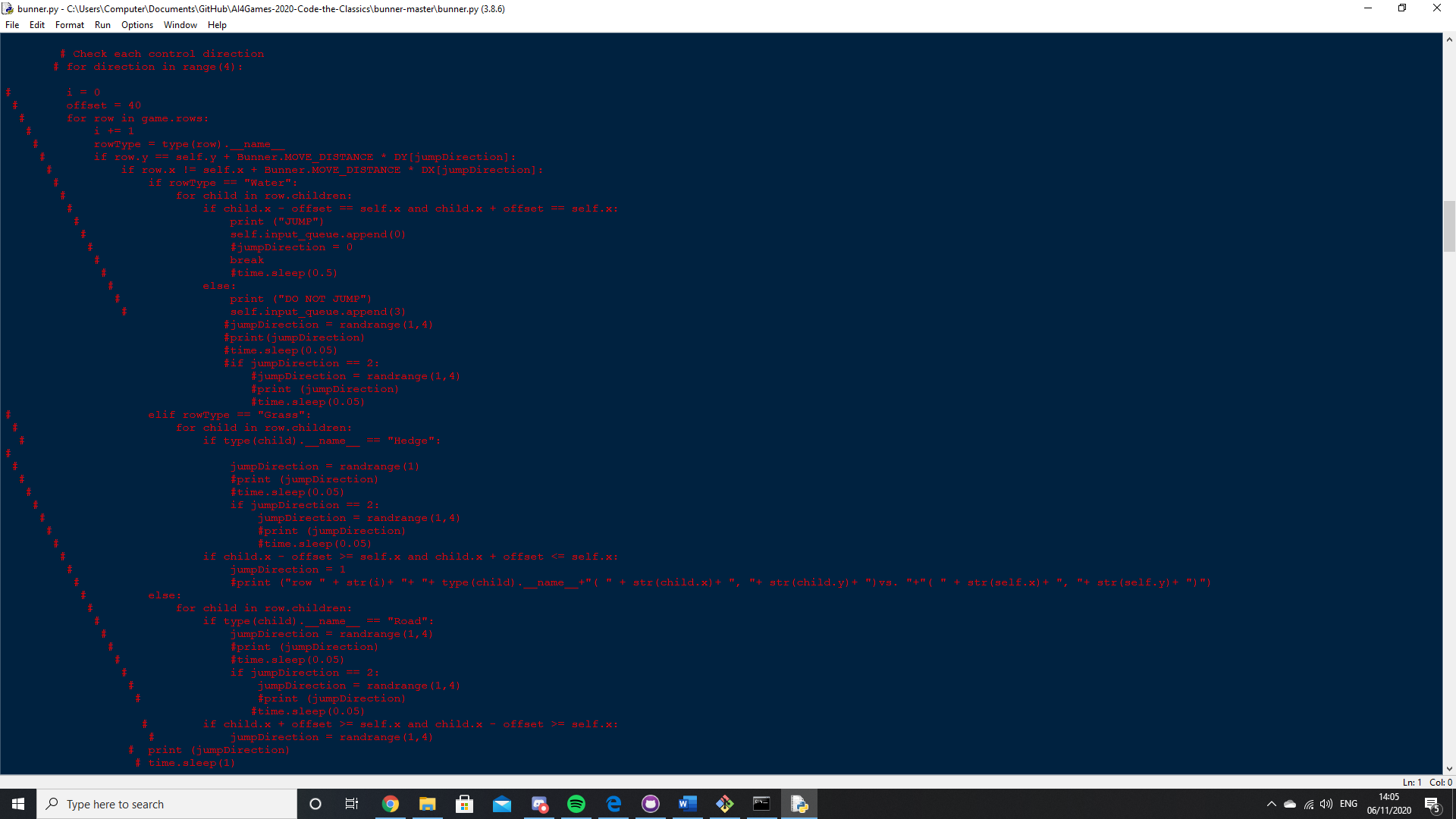
My choice

out of these two algorithms I will choose the FSM as it will allow humanization while running, so even if this game could have been made to be a competitive game then the actual players could have a chance with getting the high score.

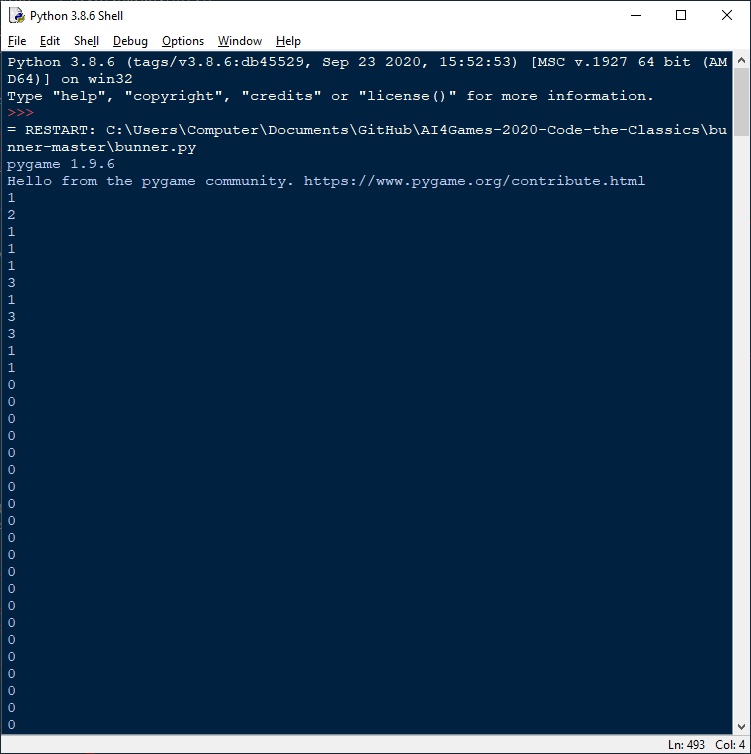
I have made a finite state machine for the Bunner game, so I can get the general idea for what id need to make when it comes to the programme.

Testing

from my code I went through different stages on how I would try and implement the given task. For the start I tried to only register the row that was in front of it and register all the children in that row for the bunny to try to avoid.

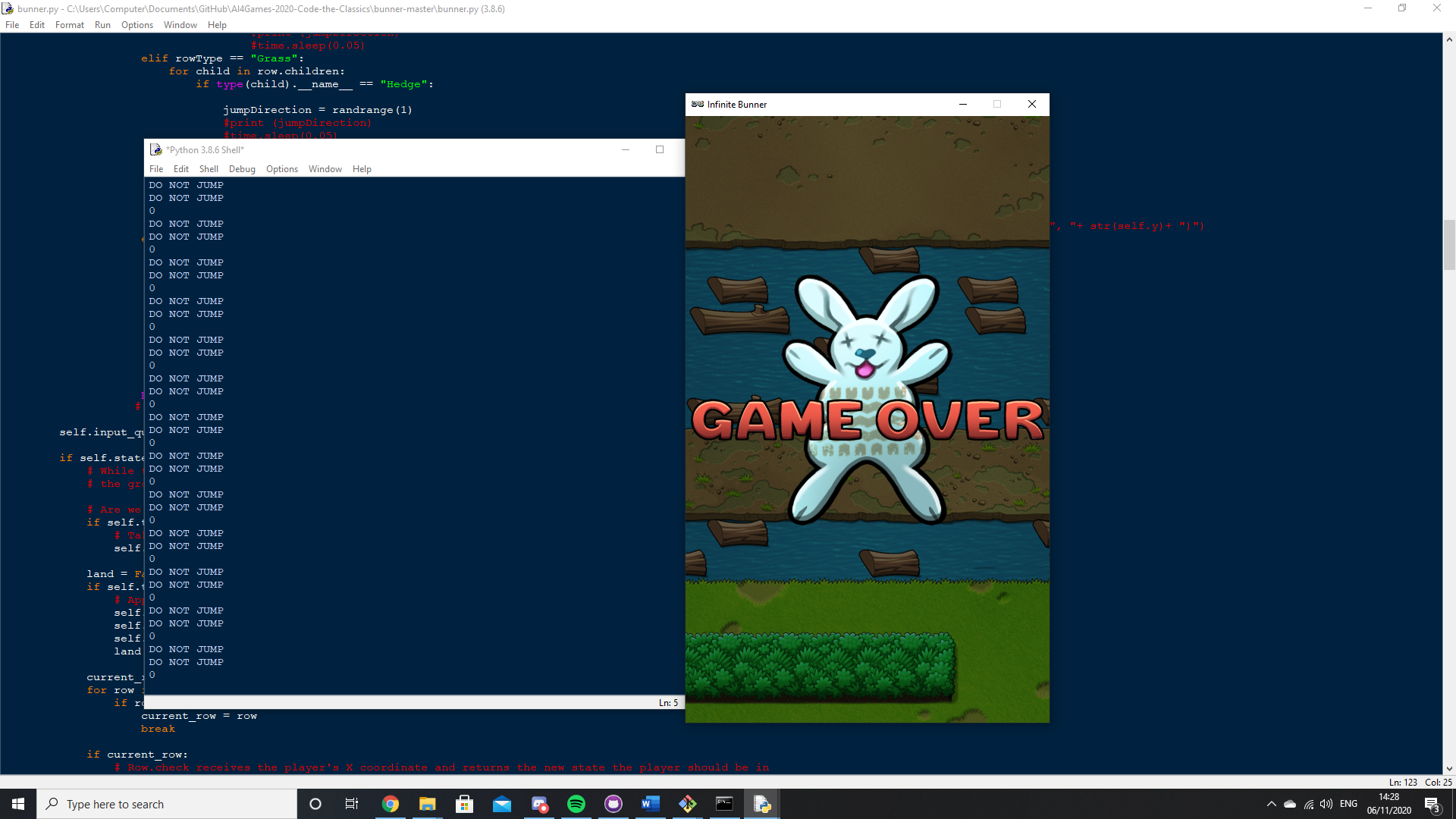


These are the logs for that given code, explaining where the bunny is going, in this code I made the bunny move randomly until it was going to hit something then it would try to move forward

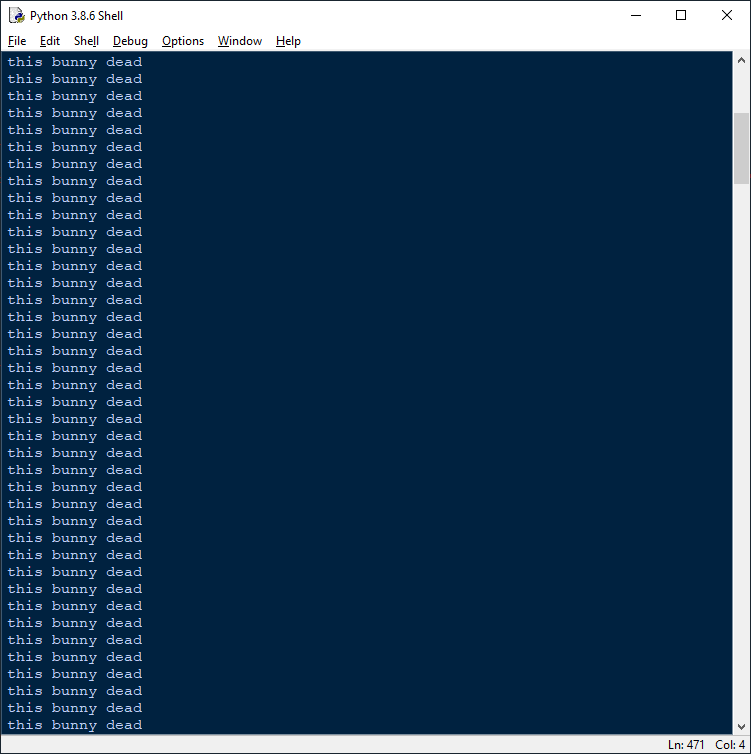


But this then caused an error as it would just get stuck on hedges and not try to move to the left or right to find an opening.

It was the same for using water, so it would check if there was a water row in front of it and then if it were to collide with a log it would output “Jump” so it will jump onto the log, but the error with this is that it would never move onto the log until the log was not in front of the bunny any more



So, this is when I decided to look into mikes demo and take inspiration from that, to check the next state of the bunny before it moves, and if it is in a death state then don’t move forward.



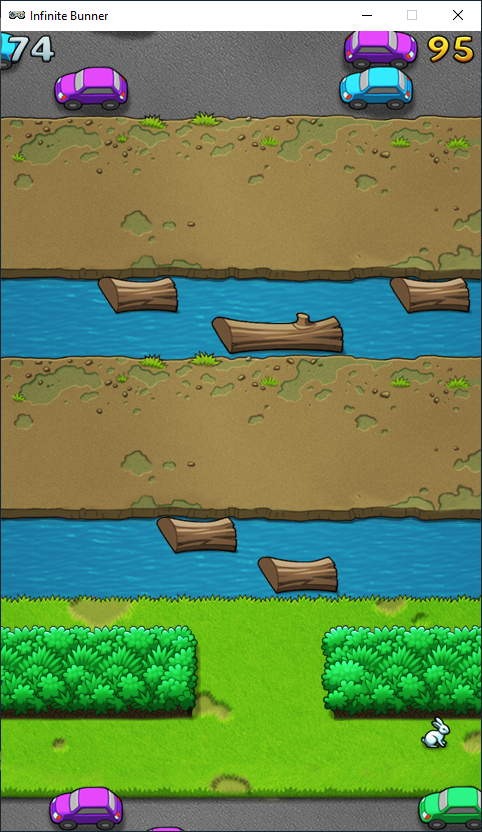
This then output what its next state was this was next to water without a log in front on the bunny



The log does not output anything if the bunny can make the move.

This also does the same thing when it come into contact with cars and trains.

I have used my old technique to check if there was a hedge in front of the bunny and then from there had a loop to go to the left and check collision again for the bunny to go up, if nothing is collided when it gets to the side of the screen then it will go the other way to find the opening.



this screenshot shows that the bunny will go to the left because its facing that way, it didn’t see the opening in the hedge as it was on logs beforehand which took it further than the opening.

Code

current\_found = False

current\_row=None

next\_row = None

PrevY = 0

behindHedge = False

hitRight = False

for row in game.rows:

if current\_found:

next\_row = row

break

if row.y==self.y:

current\_row = row

current\_found = True

if next\_row:

NextStatex = next\_row.check\_collision(self.x)

#print(NextState)

PlayerNextState = str(NextStatex)

if str(NextStatex) == "SPLAT":

print("x: " + NextStatex)

if(PlayerNextState.find("ALIVE") == -1):

print("this bunny dead")

else:

rowType = type(next\_row).\_\_name\_\_

prevY = self.y

#test for the very right of the screen

# for i in range(16):

# self.input\_queue.append(1)

# print("x: " + str(self.x))

#### ALL OF THE PRINT FUNCTIONS THAT ARE COMMENTED OUT WERE FOR TESTING PURPOSES ####

#print(self.x)

#print("self.y: " + str(self.y))

#print(prevY)

self.input\_queue.append(0) #this is to move the bunny up by one

#print("self: " + str(self.y))

#print("prev: " + str(prevY))

#print(behindHedge)

if rowType == "Grass": # this is to detect is the row in fromt of bunny is grass

for child in next\_row.children: # checks if there are any children of grass in row

if next\_row.collide(self.x, 20): # checks if bunny will collide

#print(behindHedge)

behindHedge = True # makes it known that the AI is behind a bush

if behindHedge == True: # if bunny is behind bush then run these

if hitRight == False: # if the player has hit the right of the screen

for i in range(10): # run this code 10 times

self.input\_queue.append(1) # move right

#print("x: " + str(self.x))

if self.x > 400: # check is bunny is at the edge of screen

hitRight = True # bunny hit right

if next\_row.collide(self.x, 0): # checks if colliding again

continue # do the for loop again

else: # otherwise

behindHedge = False # not behind bush

self.input\_queue.clear() # clear all inputs in queue

#print("here Right")

self.input\_queue.append(1)# move right once more

self.input\_queue.append(0)#move up

break#break out of for loop so it doesnt run again

#print("x: " + str(self.x))

if hitRight == True: # checks if bunny has hit right

self.input\_queue.append(0) # try to move up

self.input\_queue.clear() # clear all inputs in queue

for i in range(10):#run this 10 times

self.input\_queue.append(3)#move bunny left

#print ("here Left")

#self.input\_queue.append(0)

if next\_row.collide(self.x, 0): # checks collision

continue # loops again

else:#otherwise

behindHedge = False # not behind bush

self.input\_queue.clear()#clear all inputs in queue

#print ("here Left")

self.input\_queue.append(3)#move left again

self.input\_queue.append(0)#move up

break#break from for loop

break #break from collision for loop

if behindHedge == False: # checks if not behind bush

self.update # updates self

else:#otherwise

break # breaks from overall loop