 **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.* |

**Artificial Intelligence for game developers assignment - Bunner**

Finite State Machine:  
A finite state machine is a model of computation for specific states, where only one state can exist at a time, but through decisions can move on to the next state, such as the state of a character in a video game standing still is a state, but there are two decisions that can be made, these being standing still or moving, if the decision to stand still is made, the state will loop back to standing still and be unchanged, however if the decision to move is made, then the state will change to movement, and be given new decisions, which can potentially loop back to standing still or do something new.

One disadvantage that comes to mind for the finite state machine is that you cannot access every state at any given time, and would have to loop through the multiple stages to get to a certain state instead, whereas the A\* search algorithm could find the path instantly.

A\* Search Algorithm:

A\* Search algorithm is an algorithm that is capable of finding and creating optimal paths when searching for something, such as the fastest route for a train to take to a specific location, it is very complex, and if it cannot find an end to its route, it will search infinitely, which is a major drawback.

A\* search algorithm would be useful for a Bunner AI, however it would be too perfect, and always find the optimal path, while this assignment required a more human-like AI to control it, therefore the finite state machine would be more valid to use, as it can be programmed to make small mistakes or errors.

FSM scenario

With this Finite State Machine scenario, it displays the ideal states it should pass through for the Bunner game, beginning with moving up, it can either see a bush, in which case it will move to the left, if it no longer sees the bush or the edge of the screen, it will return to moving up, however if it does, it will move right, which will also return to moving up upon eventually seeing no bush or the edge of screen. If while moving up it sees either a car, train or water, it will then idle, while waiting for a log to float over or car and train to pass by, where it will then continue to move up.

See no bush or edge of screen

Car/train/water in front

See bush

Clear area in front

See edge of screen and bush

See no bush or edge of screen

Implementation

The actual implementation was close, but not perfect, the bunny is capable of avoiding traffic, trains and the river, however upon meeting a bush, it will run to the side of the screen, and back to the other side if there is no way up, but then zig zag up and right when passing the bushes, eventually falling into whatever hazard may be in front of it at the time, returning to the original ‘’move up’’ loop was difficult and did not get implemented.

Testing

A picture containing map

Description automatically generatedIdle state – shows the bunny simply waiting for a log to pass by, will take this state when a car or a train is in front of it as well. Humanlike error is included in this, as it only recognises what is directly in front of it, if a log is just passing by, the bunny will try to jump on it even if it will pass before it can finish the jump, same with cars and trains, it will try to cross the road even if a car or train is one space away, and will then be ran over.

A picture containing computer

Description automatically generated

Splat state – human error from assuming no car directly in front of it, when the car was one space to the right, which occurs for the end of logs and the start of trains as well.

A close up of a device

Description automatically generatedGraphical user interface, application

Description automatically generatedTesting – python shell lists bunny’s position at all times, what the next row is, and whether there is something in front or not, labelled as ‘’no” and “yes”, used for testing positions and space in the game. Image also shows bunny successfully crossing road.

Traversing bushes – bunny will run to the right to check for openings in the bush when meeting a bush, then to the left if no opening is found, there is an issue with this, as the bunny will continue up but zig zag up and to the left or right, based on which direction it was already going, and will not loop back to moving forward and checking for obstacles, so the bunny will fall into whatever hazard is further ahead.

Appendix

This is the code that was changed, the end of it is commented out as it was code that either didn’t work, was not implemented, or not required from the start.

def update(self):

current\_found = False # Putting in a flag, so we can run once more through the for loop

current\_row = None

next\_row = None # We need a new variable to store the next row data

currentY = None

PrevY = None

for row in game.rows:

if current\_found:

next\_row = row

print('Next row.y: ' + str(row.y))

break

if row.y == self.y:

current\_row = row # \*\*\* Here is where you could also set next\_row to do look ahead stuff

current\_found = True

print('Current row.y: ' + str(row.y))

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

if next\_row:

suggested\_state, suggested\_obj\_y\_offset = next\_row.check\_collision(self.x)

test = str(suggested\_state)

if str(suggested\_state) == "SPLAT":

print('State: ' +str(suggested\_state) + ' Y Offset: ' + str(suggested\_obj\_y\_offset))

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

print("NO")

else:

print("YES")

prevY = currentY

currentY = self.y

if self.x > 400:

self.input\_queue.append(DIRECTION\_LEFT)

elif self.x == 40:

self.input\_queue.append(DIRECTION\_RIGHT)

elif rowType == "Grass":

for child in next\_row.children:

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

self.input\_queue.append(DIRECTION\_RIGHT)

currentY = self.y

if self.y > currentY:

break

self.input\_queue.append(DIRECTION\_UP)

# i = 0

# for row in game.rows:

# i = i + 1

# RowStrType = str(type (row))

# RowType = type(row).\_\_name\_\_

# if row.y == self.y + Bunner.MOVE\_DISTANCE \* DY[MoveDirection]:

# if row.x != self.x + Bunner.MOVE\_DISTANCE \* DX[MoveDirection]:

# if RowType == "Water":

# for child in row.children:

# if type(child).\_\_name\_\_ == "Log":

# print("row" + str(i) + ":" + type(child).\_\_name\_\_ + "(" + str(child.x) + ", " + str(child.y) + ")")

#print ("row "+str(I)+":"+RowType)

# for theboy in row.children:

# print ("row "+str(I)+":"+str (type(theboy)))

#if row.y == self.y + Bunner.MOVE\_DISTANCE \* DY[MoveDirection]:

#if row.allow\_movement(self.x + Bunner.MOVE\_DISTANCE \* DX[MoveDirection]) is False:

#return

#if next\_row: suggested\_state, suggested\_obj\_y\_offset = next\_row.check\_collision(self.x)

#print('State: ' + str(suggested\_state) + ' Y Offset: ' + str(suggested\_obj\_y\_offset))

#sys.exit()