

Artificial Intelligence for Game Developers

Assignment 1: Creating the Classics

Student ID: 18138284

Table of Contents

[Table of Figures 4](#_Toc55591431)

[1 Introduction 5](#_Toc55591432)

[2 Modern AI & AI Theory 8](#_Toc55591433)

[3 Gameplay Fundamentals 10](#_Toc55591434)

[4 Understanding the Source Code & Structure 13](#_Toc55591435)

[5 Creating the AI 15](#_Toc55591436)

[5.1 Iteration 1 (Random Movement Bunny) 15](#_Toc55591437)

[5.2 Iteration 2 (Future Reading Bunny) 17](#_Toc55591438)

[5.3 Iteration 3 (Smarter Thinking Bunny) 18](#_Toc55591439)

[5.4 Iteration 4 (Final Bunny) 19](#_Toc55591440)

[6 Programming Tools 21](#_Toc55591441)

[7 Improving the AI Further 22](#_Toc55591442)

[8 Conclusion 23](#_Toc55591443)

[9 Code Appendix 24](#_Toc55591444)

[References 31](#_Toc55591445)

# Table of Figures

Figure 1 – A book illustration of the fake chess playing AI: The Turk 5

Figure 2 – In Red Dead Redemption 2, the player can play poker hands with NPCs who will remark on their luck and sometimes even tell personal stories about themselves. 5

Figure 3- An example of an FSM model for a shooter game. This AI has several states associated with it based on interactions with the player. If the player kills the AI (no health left) the Deceased state is activated, and the AI has no more control over the game world (Santatan, 2020). 7

Figure 4- An example of the A\* Search Algorithm (Belwariar, 2018) 8

Figure 5 – A screenshot of Infinite Bunner that shows some of the many hazards the player will need to transverse. 9

Figure 6- A Finite State Machine (FSM) design created for an Infinite Bunner AI 10

Figure 7 - Infinite Bunner code showing the MyActor base class that others inherit from 11

Figure 8- The update function within the Bunner class already has controls for the player that can be replaced for the AI 11

Figure 9- In the Water class, there is a 50/ 50 chance the next row will be more water or dirt. This means that there is a randomness to the game, but the logic can be formulated into a strategy. 12

Figure 10- Example code in the Road class showing how cars can have random velocities applied to them. 12

Figure 11- Infinite Bunner original code showing how the screen scrolls faster based on the player's position. 12

Figure 12- Each directional control is given a random value and will execute every time the update function is called 13

Figure 13- The bunny will now only move in a direction if a 1 in 30 chance has been achieved. This number was chosen with experimentation (less than 30 was moving the bunny too much, over 30 would move it too little). 14

Figure 14- Original Infinite Bunner code which is used to check if the bunny has died 14

Figure 15- AI code used to check bunny collisions. The find function is used for Python strings to detect if a word appears in the specified string, which will determine the selected option for the AI (Dharmkar, 2017). 14

Figure 16- As Grass, Dirt and Pavement tend to be safe spots, all the AI will need to do is move up (unless there is a hedge on the Grass sections). Road and Rail have similar properties, so they are grouped together along with Water which has its own statement as well. 15

Figure 17- The top image shows the many positions used to check against the AI’s current position. The bottom image shows how some of these variables are checked together so that the bunny does not run into danger quite as often. 15

Figure 18- The code will only run if the waitInt is equal to 1 which gives the controller output some relief. 16

Figure 19- Hedge navigation code. 16

Figure 20- Centre hedge navigation for if the AI is stuck, success can vary. 17

Figure 21- Screenshot of the Console outputting AI messages. 17

# 1 Introduction

For hundreds of years, scholars and scientists alike have theorised the day that humanity would be able to replicate the actions of human beings mechanically. Nowadays, artificial intelligence (AI) is used in many professions, businesses, and technologies for all manner of work, but replicating certain actions of humans is a near impossible task.

Science fiction dreams of a future filled with androids and complex computer systems that are tasked with easily computing actions like that of a human brain, but could such a future ever become a reality? Ager theorises that computers replicating a human’s brain will never be possible as a human’s DNA and structure are far more complex than the inner workings of a computer system (Ager, 2017).

The Turk was an 18th century chess playing “machine” designed to beat any human at the game. Inside were many gears and mechanisms which could be shown to audiences to convince them that they were truly watching a self-aware being play the game. In reality, it was an elaborate hoax that contained a human player controlling the actions of the puppet disguised as an android (Figure 1). Buchanan states in “A Very Brief History of Artificial Intelligence” that Samuel L. Clemens wrote in a newspaper column stating that The Turk must have been a machine because it played so well (Buchanan, 2006).



Figure – A book illustration of the fake chess playing AI: The Turk

This is an interesting quote, as an AI that can perform well at a task is difficult to make but creating an AI that acts like a human being can be even more complicated and for certain applications, near impossible.

One of the most common uses for AI is within video games. As video game graphics become more realistic and closer to reality, designers and programmers are faced with the possibility that modern AI will not be able to keep up with the realism of the worlds players will be put into.

Immersion in games is very important, especially for triple-A titles such as Rockstar’s 2018 open world Western, Red Dead Redemption 2. This game is considered by some to be one of the most realistic games of the modern era, in part thanks to its superb attention to the non-player character (NPC) AI.



Figure – In Red Dead Redemption 2, the player can play poker hands with NPCs who will remark on their luck and sometimes even tell personal stories about themselves.

NPCs will react to hundreds of scenarios the player puts themselves into, and is said to contain 500,000 lines of dialogue, all of which were motion captured by 500 actors with an additional 700 voice actors (Wood, 2018). NPC events will even appear randomly whilst the player is exploring the world and can often have quest lines attached with unique dialogue choices.

With Rockstar’s hard work, the game is well polished, but by no means perfect. A player who is being robbed by an NPC during a random event who is able to tie up the criminal and take them into a nearby town to be prosecuted may be surprised to find inhabitant NPCs of the town screaming and attempting to attack the player for tying up and kidnapping a man. The player does not have the option to explain the events that transpired before if it has not been programmed into the game.

It is never the fault of a player to attempt actions outside the boundaries of a game. Video games are designed to test limits and simulate life in a safe and non-consequential environment. As Red Dead Redemption 2 proves though, no matter how many people and resources you throw at a game, there perhaps will never be an AI that is truly perfect.

Imitating human life has too many variables to consider. So instead, the programming of AI in games is often focused more on fairness and enjoyment. If an AI is too good at a game, it is not particularly fun for the player and does not feel as realistic as it is human nature to make mistakes.

Capcom’s 1991 arcade fighting game, Street Fighter II: The World Warrior featured notorious AI that would cheat against human players by mashing buttons faster than a human ever could (sf2platinum, 2017), (Good, 2019). Poor AI implementation is not just limited to competitive games. In Mossmouth’s 2020 platformer Spelunky 2, hired hands are NPCs designed to aid the player with their adventures and have the same abilities the player has. As the hired hands are designed to follow the player at proximity, if the player tries to use their whip ability too close to the hired hand and too often, the hired hand will think that the player is trying to kill them and will whip them to death, resulting in a Game Over state.

These examples outline the challenges and issues involved in AI development. Ultimately, it is up to the programmers to define what is fair and balanced through testing the AI rigorously, but it is just as important that they understand the design and functionality of the game.

For this assignment, we have been tasked with creating an AI for one of four arcade style Python games. This report will outline the AI theories, challenges, code, and implementation used to create an AI.

# 2 Modern AI & AI Theory

There are many techniques and algorithms used in modern day AI to make an AI preform actions. One of the most used and effective techniques is known as Machine Learning (ML), where the AI is given an overall goal and will experiment each time it is ran to learn more about the game world. Eventually, through repetition the AI will know the exact features of the game that will help it achieve its set goal and will be able to preform the goal with great efficiency.

A finite state machine (FSM) is one of the simplest and most used AI algorithms for games. The AI is given a state it will always begin at and based on certain conditions the AI will pick the next state it needs to go into. These states will either loop indefinitely or until a final state is reached (Figure 3).

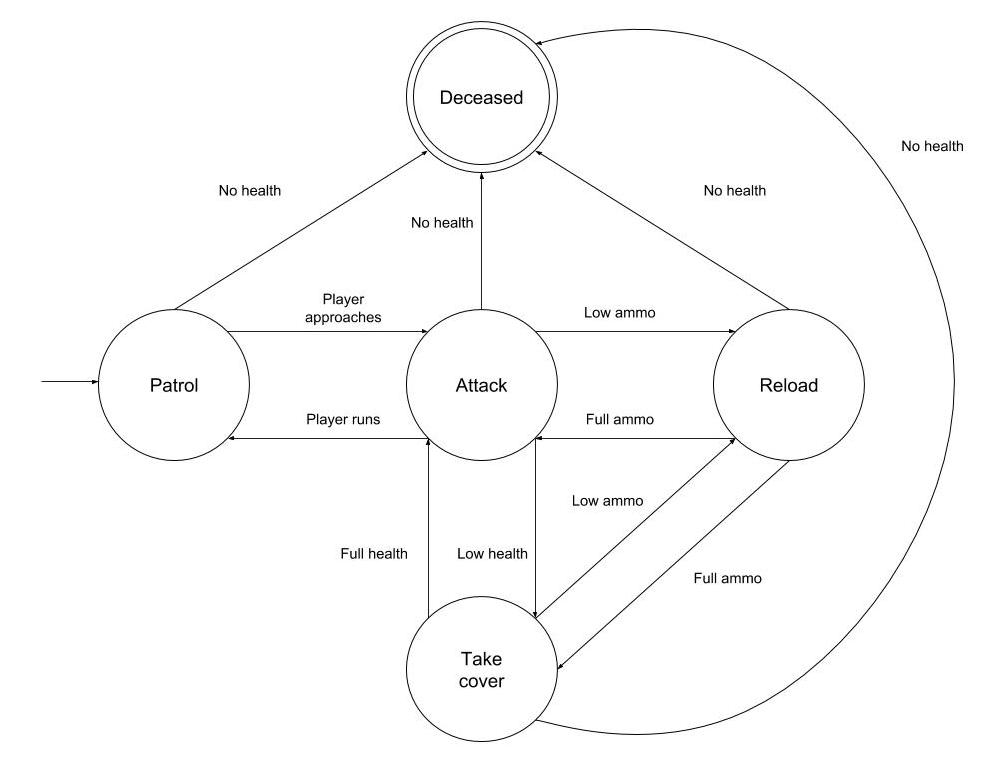


Figure - An example of an FSM model for a shooter game. This AI has several states associated with it based on interactions with the player. If the player kills the AI (no health left) the Deceased state is activated, and the AI has no more control over the game world (Santatan, 2020).

Pathfinding is another important algorithm for video game AI. In a 2D or 3D space an AI needs a way to navigate the world and reach specific points (the player for example), all whilst avoiding walls and objects that could be in its way. This can be done using pathfinding by plotting the AI’s points of interest and then figuring out the route it will take to get to it, based on the current command or state.



Figure - An example of the A\* Search Algorithm (Belwariar, 2018).

One aspect of AI that was touched on briefly in the Introduction section is whether the AI should have access to information that the player does not also have access to. As an AI is based on the game code and therefore, must receive feedback based on the game world, the AI could technically access any part of the game if it were programmed this way.

If an AI was being programmed to find the fastest route in a maze, what would be the point of going all the way round walls if it could just clip straight through the walls and reach the end point much faster? This is where logic and fairness need to be considered.

For example, if the player does not have access to an enemy NPC’s health via their on screen heads up display (HUD), but the AI can see that the player is low on health in the game code and becomes more aggressive to finish them off, it is not particularly fair for the player unless the game is specifically designed to be difficult. Instead, a programmer could opt to allow the NPC to be more aggressive based on other conditions the player can see on the screen, through random chance or perhaps if the AI itself is close to being defeated.

# 3 Gameplay Fundamentals

To understand how to create an AI for any game, the rules, boundaries, and controls must be clear to the programmers involved. This will usually be discussed with the game designer and coordinated with other non-AI programmers. The AI programmers will need to have a very good understanding of what their AI is supposed to achieve within the game world all whilst considering the player’s control restrictions and abilities.

Infinite Bunner was the game chosen for this report, which is a reimagining or “clone” of Konami’s 1981 arcade classic, Frogger (Crookes, Gillett, & Upton, 2019). The player takes control of a bunny who is trying to cross a busy street. Although the game has no end state, it will continue to create new obstacles for the player to traverse until the bunny dies. The incentive for the player is their score which will increase based on the distance they can get the bunny. If they stand still for too long, the screen will scroll up and if the bunny goes off the screen this will also result in a Game Over state.



Figure – A screenshot of Infinite Bunner that shows some of the many hazards the player will need to transverse.

The hazards in Infinite Bunner are:

**Logs:** This section of the game requires the player to traverse floating logs on a river that can move from the left or right-hand side of the screen. The timing on this can be tricky, as if the player moves to a log with imprecise timing, the bunny will fall in the water resulting in a Game Over. This section is unique as players may have to go back across another log if a log in front has not scrolled passed the log the player is waiting on. Waiting on a log will eventually result in the death of the bunny as it will fall into the water when the log rolls off screen.

**Roads:** Roads can come in many different sizes, some roads will only have two lanes, but some will have multiple lanes of cars that will run over the bunny. Speed and timing are needed for this section as the patterns of the cars can often mean quite a tight squeeze for the player to manoeuvre. The player does have the option to go back if they feel a car in front will crush them, but this is risky if another car is moving in the lane behind. The cars can be moving at different speeds, some go faster than others.

**Dirt/ Pavement**: These areas do not have any hazards and give the player some time to consider their moves before attempting the next section.

**Hedges**: Hedges cannot be moved over so the player will need to go around them. This can limit the player’s position for the next area as they cannot wait and allow the screen to scroll up, ending the game.

**Train Tracks:** The train track sections are like the cars section but there is only one train to deal with that is much longer than the cars and moves much faster.

The more inputs an AI has, the more complex the AI programming model can become. Thankfully, the bunny only has 4 keyboard inputs making the game very simple (move up, down left or right with WASD on the keyboard).

Frequent playtesting was required for this AI to understand how a player would think and tackle the terrain. Data could also be collected through seeing how other players of all ages and levels of gaming experience handled the game and asking for their strategies as well as the parts they found the most difficult.

Because Infinite Bunner is a single player game it may seem pointless to create an AI that plays the game exactly the same way a human does, but there are many advantages to creating an AI for this purpose, including:

**Experimentation/ Strategy**: A perfect Infinite Bunner AI could theoretically show the programmers exploits in the game’s design/ code that human testers could not.

**Demos**: Demo screens are used in games (usually in the menus) and often showcase an AI playing through parts of the game.

**Multiplayer**: A new mode could be worked into the game where the player has to get further than the AI or survive for longer.

From going through Infinite Bunner’s source code, a simple FSM design was created that could be applied to an AI that plays the game. In Figure 6 you can see the AI will need to scan for objects within the game world and move only if it is safe to do so. Other elements may need to be taken into consideration with this model though, including whether a hazard will kill the AI from another position.

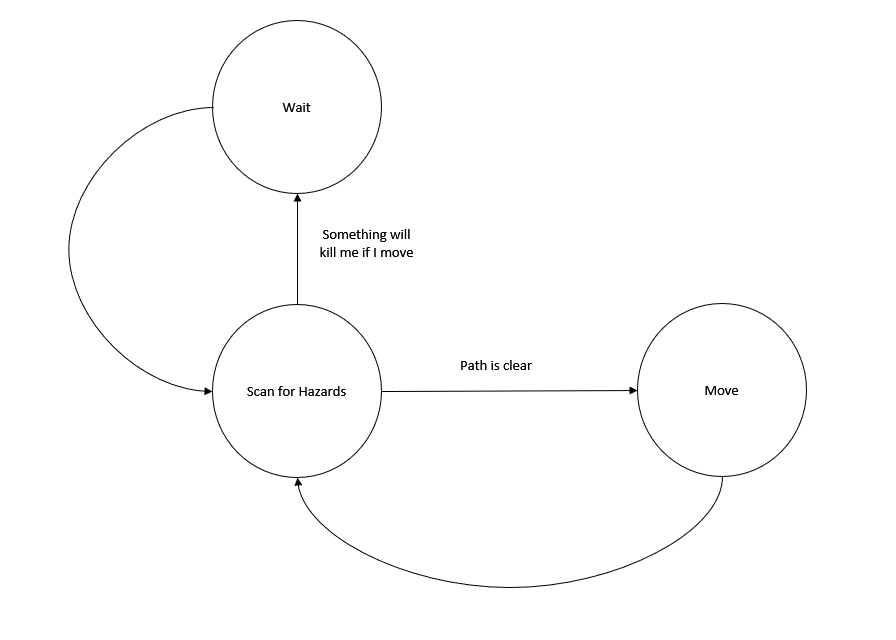


Figure - A Finite State Machine (FSM) design created for an Infinite Bunner AI.

# 4 Understanding the Source Code & Structure

To create an AI for any game, the programmers must incorporate elements of existing code into their solutions. Plenty of information will be needed including but not limited to player information, world information, objects the AI can interact with, animations, etc. Much of this information can often be derived from other classes using polymorphism.

Polymorphism and inheritance are used in computer programming to allow a class to access certain features and variables from a base class. For example, in Infinite Bunner the class MyActor is used as a base class for the player controls and the eagle NPC (Figure).

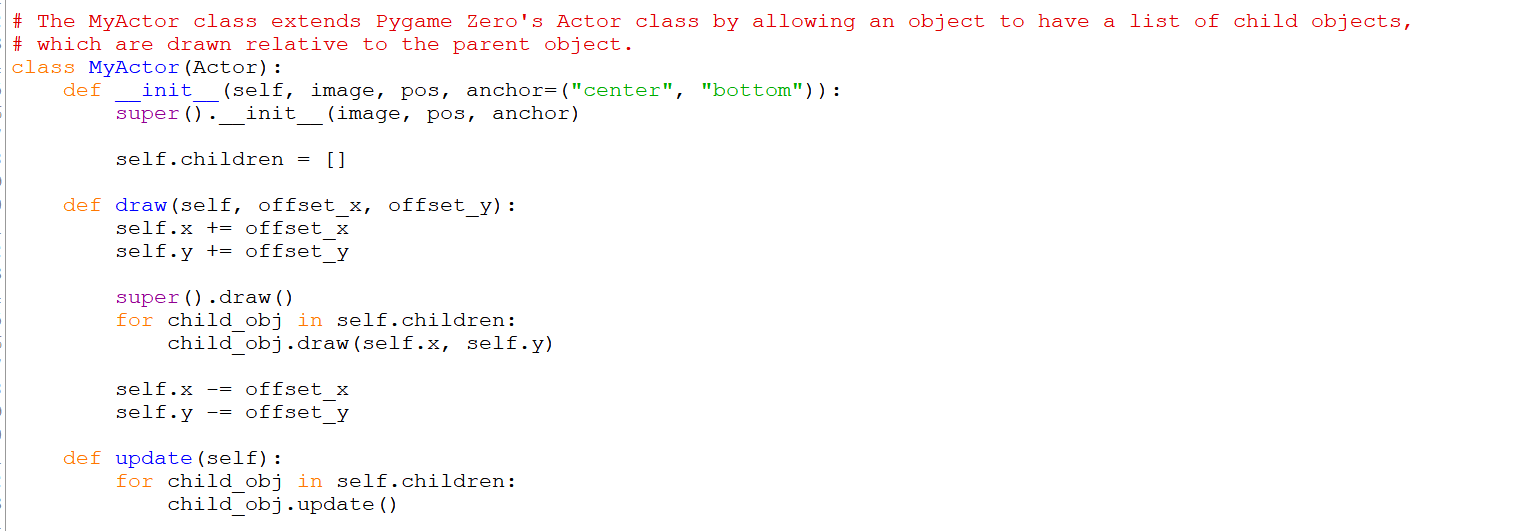
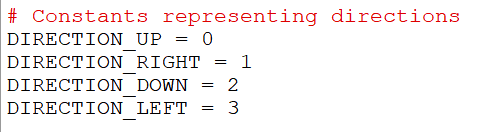


Figure - Infinite Bunner code showing the MyActor base class that others inherit from.

As the task for this assignment was to create an AI to replace the player character, there was no need to create a new class. All the controls needed for the AI could be accessed from the Bunner class.



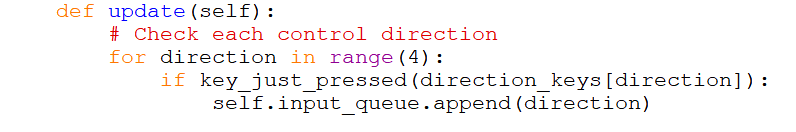


Figure - The update function within the Bunner class already has controls for the player that can be replaced for the AI.

Another aspect to consider is whether the game is random. If the game will always play the same way every time it is ran then the AI will always behave the same as well. To get round this issue programmers use seeds which change attributes of the AI based on random numbers. If the game is random then this is not as big an issue as the AI will always have new environments and sequences of events to experiment with.

For Infinite Bunner the original programmers opted for the rows that are generated to be pseudo-random, meaning although the bunny will always start the game in a safe position, the hazards and rows will be generated through a sequence of rules (Figure 9). For example, grass rows can sometimes spawn maze-like hedges using the generate\_hedge\_mask function.

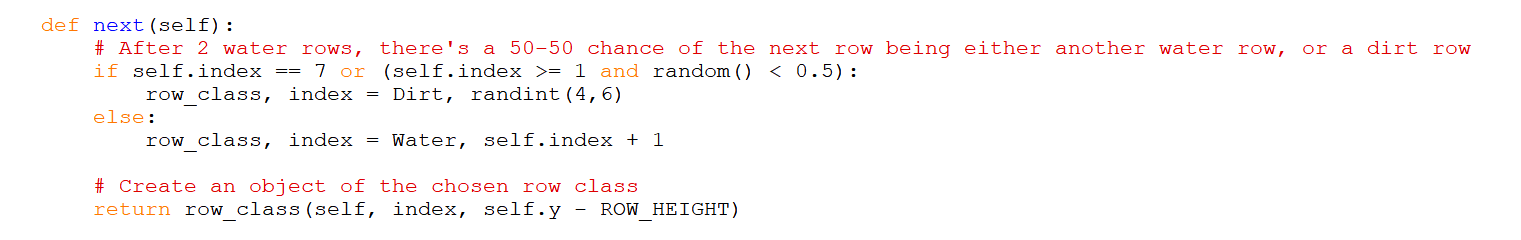


Figure - In the Water class, there is a 50/ 50 chance the next row will be more water or dirt. This means that there is a randomness to the game, but the logic can be formulated into a strategy.

The pseudo-random elements are not just limited to the types of rows being generated. The Mover class which includes objects that collide with the bunny (logs, cars, and trains) also use them. These random values are used to determine which direction the object will scroll onto screen from, as well as what speed they will be traveling at (Figure 10).

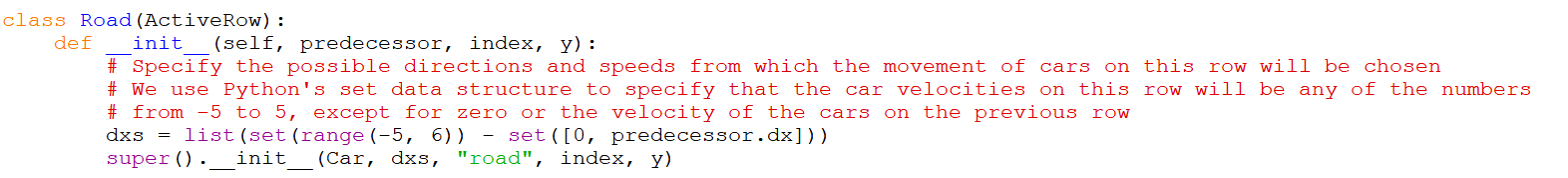


Figure - Example code in the Road class showing how cars can have random velocities applied to them.

Another thing to consider with an AI in Infinite Bunner is the fact that the screen will scroll faster if the bunny is close to the top of the screen (Figure 11).

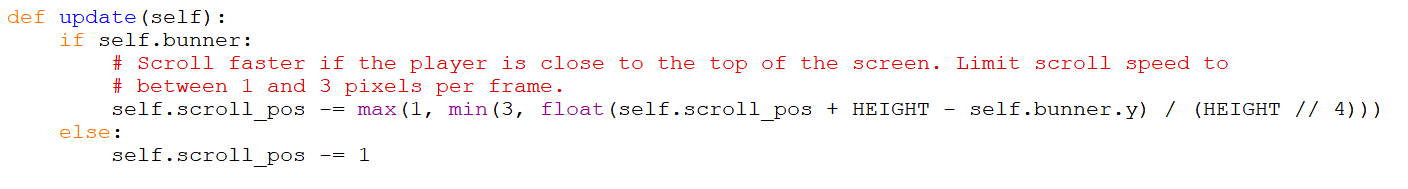


Figure - Infinite Bunner original code showing how the screen scrolls faster based on the player's position.

# 5 Creating the AI

## 5.1 Iteration 1 (Random Movement Bunny)

For iteration 1 of the Infinite Bunner AI, the main task was simply getting the AI to move by itself. As mentioned in the Understanding the Source Code section, player controls can be replaced with the AI’s movement. The random function is used to determine which direction the bunny will move. However, this is not a very effective strategy as the bunny has no chance getting through the complex levels moving completely randomly every time (Figure 12).

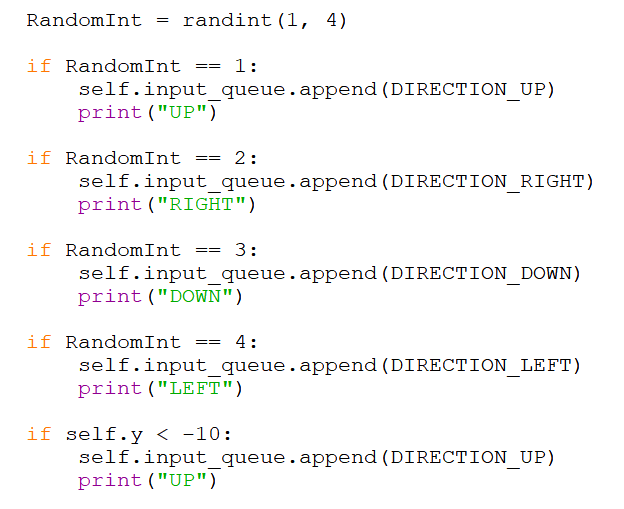


Figure - Each directional control is given a random value and will execute every time the update function is called

To add a smaller amount of logic to the if statements, the 3rd statement was removed which makes the bunny go down. Although the bunny will need to go down at times to avoid hazards, during this stage of the development going backwards did not provide any significant improvement to the AI’s gameplay.

The other problem with this method is that the bunny would always run straight into hazards. A way to get the bunny to wait was vital in future iterations or else the bunny would never stop moving. This was done using another random if statement that makes the likelihood of the bunny moving far less frequently (Figure 13).

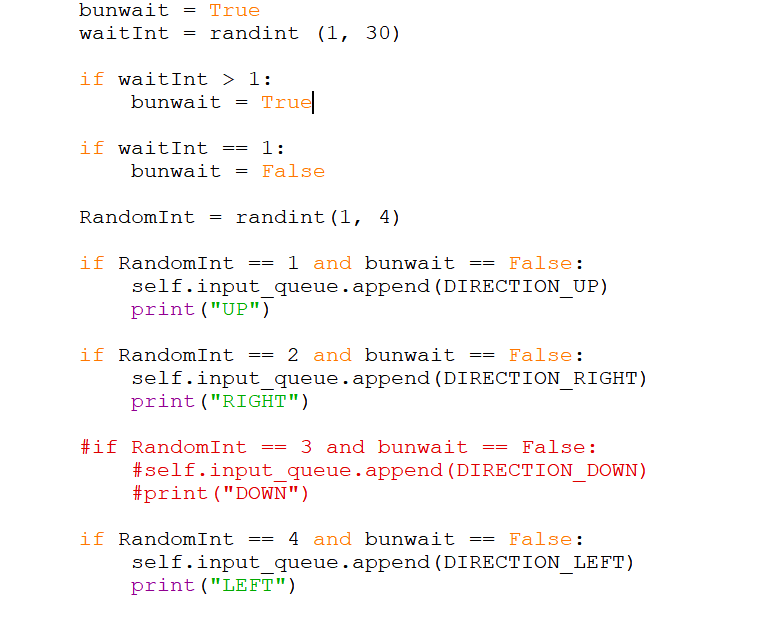


Figure - The bunny will now only move in a direction if a 1 in 30 chance has been achieved. This number was chosen with experimentation (less than 30 was moving the bunny too much, over 30 would move it too little).

## 5.2 Iteration 2 (Future Reading Bunny)

Now that the basic controls have been implemented for the AI to work, refinement of the code is needed to apply some logic. The AI needs to understand details about the game world including whether it will die if it moves forward. The original Infinite Bunner game code already has a function for checking to see if the bunny has collided with another object called check\_collision (Figure 14).

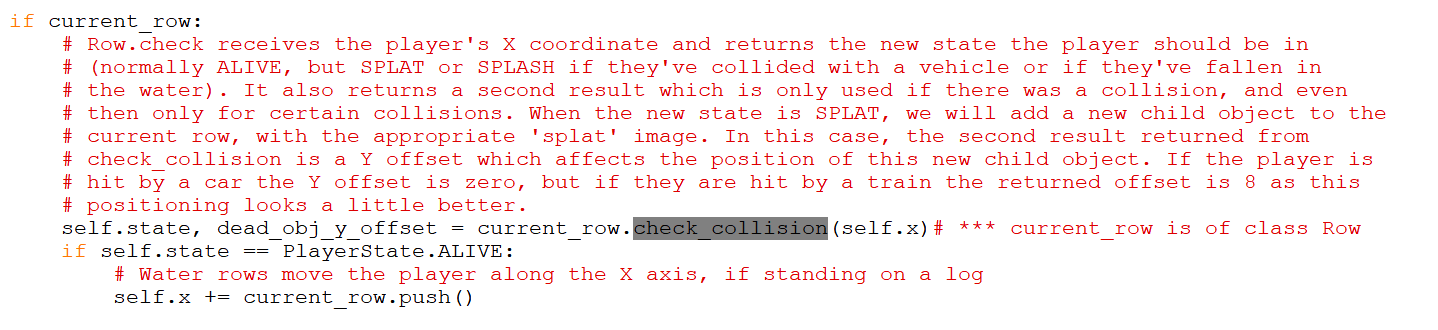


Figure - Original Infinite Bunner code which is used to check if the bunny has died

The final version of this iteration is quite simple, as it checks the row in front of the bunny based on its X coordinates and uses the collision test to detect if its going to die or not. If it is going to, it will do nothing. If the path seems clear it will move up (Figure 15).

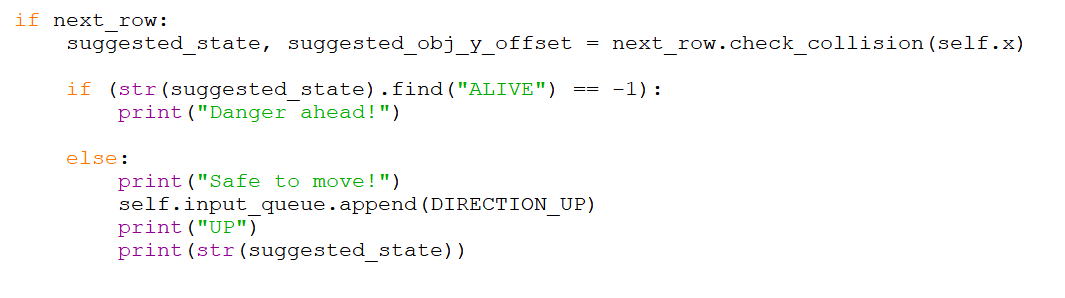


Figure - AI code used to check bunny collisions. The find function is used for Python strings to detect if a word appears in the specified string, which will determine the selected option for the AI (Dharmkar, 2017).

The AI can now stop and wait for any potential deaths that may occur ahead of it. The limitations of this iteration are the fact that all the AI can do is move forward and wait. It cannot detect if cars are going to hit it from the side and because a car can still move into a safe spot, it can detect the path is clear but still get hit once it moves. The same issue occurs on the logs, where it may think it can jump onto a log in front of it, but the log will move away and the bunny will fall into the water.

## 5.3 Iteration 3 (Smarter Thinking Bunny)

With this iteration, many new conditions were placed on top of existing conditions to refine arguments based on the AI’s surroundings. As different patches of land have different properties, each set of statements is split up, so the AI has to check what kind of row it is dealing with (Figure 16).



Figure - As Grass, Dirt and Pavement tend to be safe spots, all the AI will need to do is move up (unless there is a hedge on the Grass sections). Road and Rail have similar properties, so they are grouped together along with Water which has its own statement as well.

For the road section, the AI was constantly getting crushed by cars whilst waiting for the car in its row ahead to move. In Figure 17 you can see that it now has the ability to check if there is a car coming from the side and will attempt to move in the opposite direction of the traffic and move up if there is nothing ahead of it.

The if statement used in Figure 17 for checking the next row is clear is now quite long. This is because the AI not only checks the row directly in front of its position is clear, but it will also check that the adjacent left and right positions of the following row is clear as well. This was added to stop the bunny running up a row too early which effectively works most of the time, but sometimes it will still move too soon.

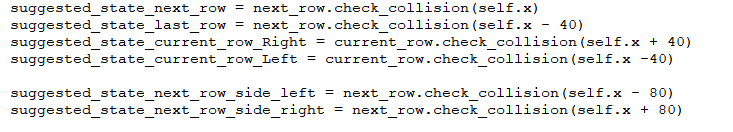
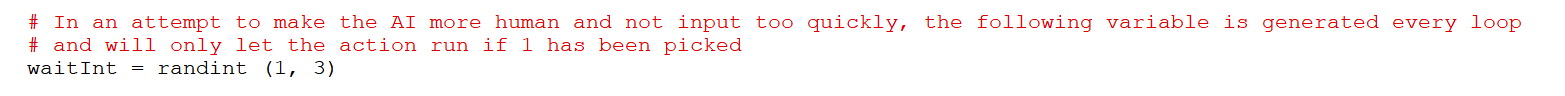


Figure - The top image shows the many positions used to check against the AI’s current position. The bottom image shows how some of these variables are checked together so that the bunny does not run into danger quite as often.

## 5.4 Iteration 4 (Final Bunny)

As the AI was outputting controls too quickly, a random integer is used to control how quickly the AI can make movements (Figure 18).



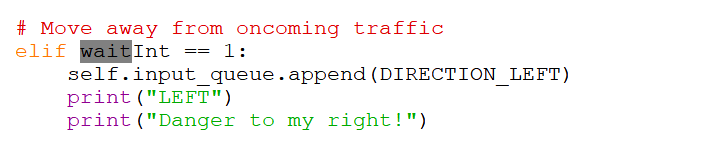


Figure - The code will only run if the waitInt is equal to 1 which gives the controller output some relief.

An attempt was made to get the AI navigating hedges but as the accessing of child objects such as hedges could not be figured out, another solution was used that is not very effective. A variable is set increase every time the AI attempts to go up and will execute the maze controls if it has reached 10 attempts without doing something else. When 10 attempts are made, it will figure out which side of the screen it is closet to and run to the opposite direction until a way out is found (Figure 19).

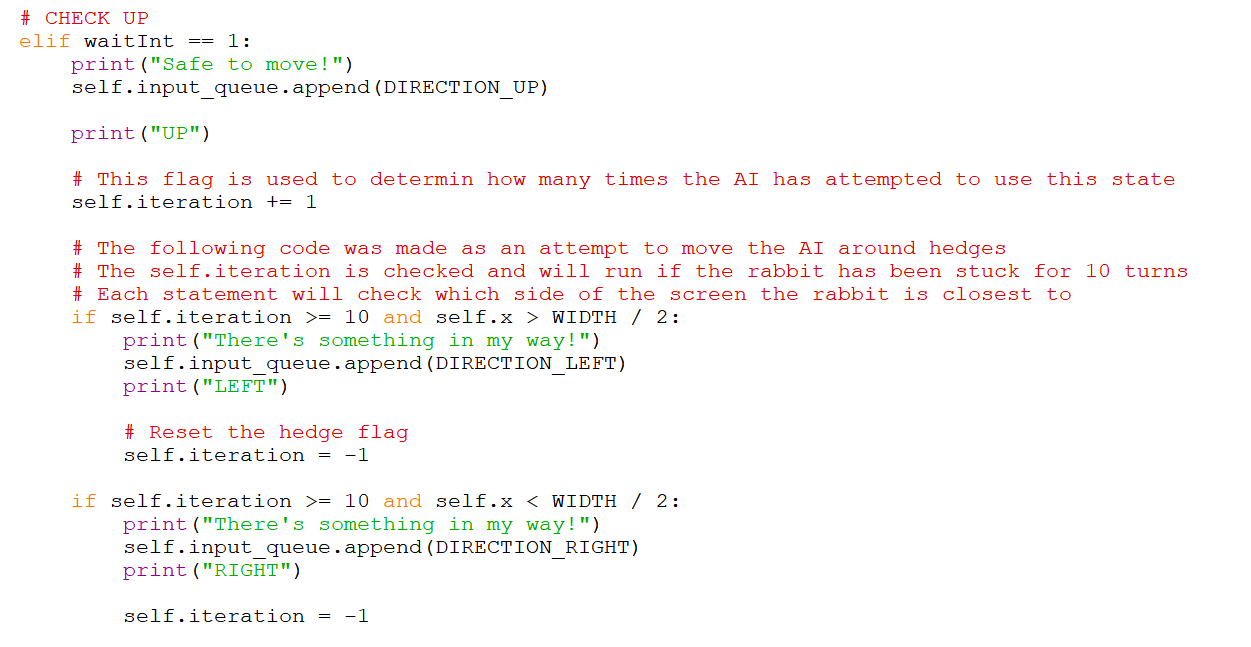


Figure - Hedge navigation code.

This does not always work correctly as sometimes hedges can spawn in the centre of the screen which is why the code in Figure 20 was attempted. It is designed to check if the middle location is occupied by the bunny and will then pick a random direction to move (left or right) for a random number of iterations. The theory was by picking a random path the AI would eventually make it to an exit but a more logical method is likely needed where the AI can actually detect the hedge itself if the AI was to be improved.

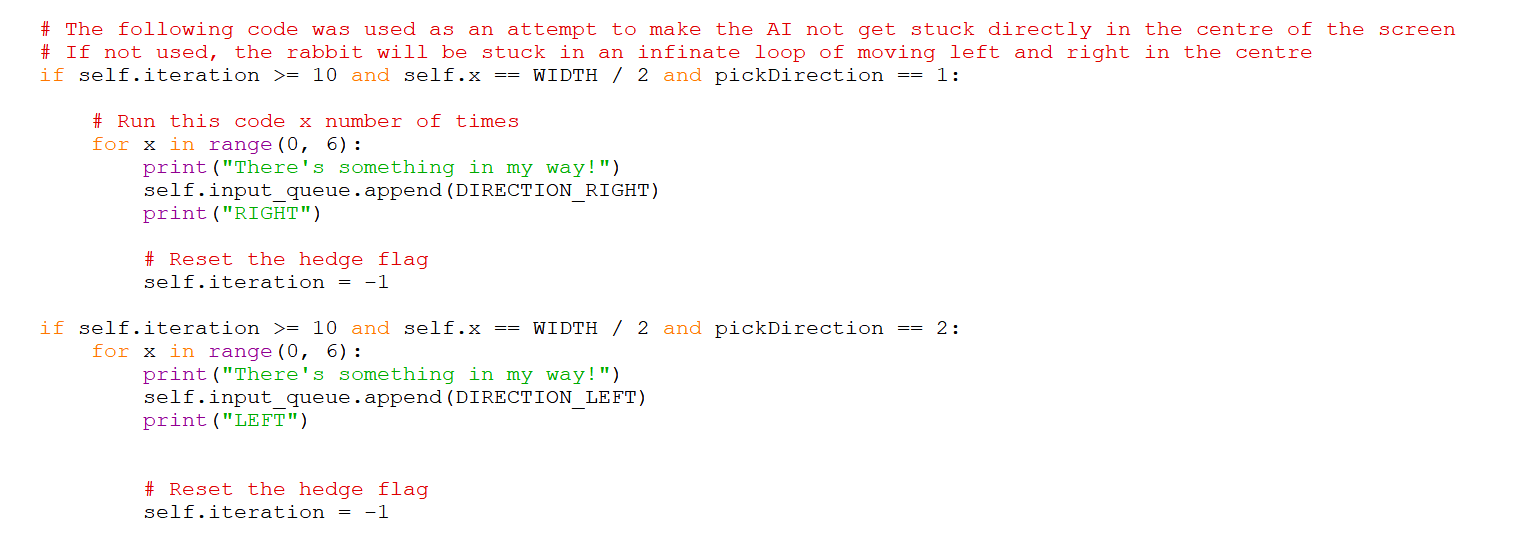


Figure - Centre hedge navigation for if the AI is stuck, success can vary.

Allowing the checking for cars on the left and right hand side of the following row was implemented in the previous iteration, it has been removed from this iteration for the Grass section of code and is only applied during the Road/ Rail sections. This is because the if statements shown in Figure 17 were conflicting with the Water section, meaning there was something wrong transitioning from Road/ Rail to Water (the AI would not move at all if in this situation).

This means that if the AI starts the game on grass it can die immediately if a road is the next row as it does not have the ability to check if it will die from the left and right hand sides of the row in front and will move to early.

# 6 Programming Tools

To create this AI, many messages are printed to Python’s console to give feedback about what the AI is thinking (Figure 21). Many other variables were outputted to the console during the development to check that the code implementations were correct such as positional data.

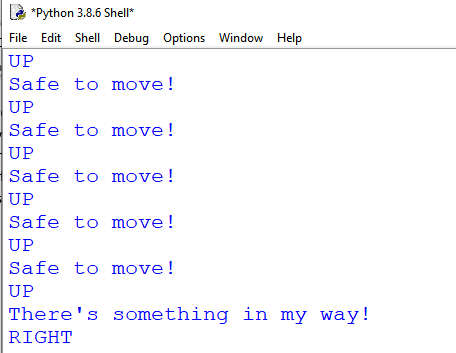


Figure - Screenshot of the Console outputting AI messages.

Many other tools could have come in useful for this assignment. For example, test environments are used in AI to check certain conditions are working correctly which are levels with only certain elements used to see how the AI will handle them. When programming the AI hedge navigation, it would have been quite useful to increase the likelihood of a hedge being generated and not have to wait for the game to create one naturally.

# 7 Improving the AI Further

The final implementation of the AI is quite flawed and has many issues. At times it will act like a human is controlling it and at other times it acts like an AI. To improve the AI further, a rethink of the code is likely needed. An A\* algorithm where the AI reads the entire map would be better.

The speed a car is going would have also been a useful thing for the AI to know as it will quite often run into the next lane, thinking its safe but really it would not have time to move to the following row before getting hit.

Another improvement could be the AI knowing how close it is to the edge of a screen. Most players will position themselves back to the centre if too close to a side as they cannot see the hazards coming from off screen until it is too late.

It also would have been nice if the AI could move to the position of a log when starting the log section, as it will waste time waiting for the log to line up with the bunny. It also does not have the ability to back track if a log is coming from behind it and no long is coming in front of it.

# 8 Conclusion

The final implementation of the Infinite Bunner AI created for this assignment is both disappointing and impressive at the same time. Some runs, the AI uses excellent agility and can get quite far. Other times it will die right away or run straight into a car. The reason for these mixed results is likely due to the way the AI has been programmed. Logically, this felt like the only way to implement the AI whilst still being able to understand what it was doing but many other logic methods would probably have worked a lot better.

# 9 Code Appendix

Please note that the following code starts in the Bunner update class for the AI and does not include the entire solution. For full code, please see the bunner-18138284 branch on GitHub.

## AI STARTS

# rowType is used to check the name of the type of row the bunny will have to navigate

rowType = type(row).\_\_name\_\_

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

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

for row in game.rows:

if current\_found:

next\_row = row

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

# Check the next row before moving

if next\_row:

# The following variables are used to check collisions at different locations of the AI

suggested\_state\_next\_row = next\_row.check\_collision(self.x)

suggested\_state\_last\_row = next\_row.check\_collision(self.x - 40)

# Checks the current rows left and right hand side for collisions

suggested\_state\_current\_row\_Right = current\_row.check\_collision(self.x + 40)

suggested\_state\_current\_row\_Left = current\_row.check\_collision(self.x -40)

suggested\_state\_current\_row\_Right2 = current\_row.check\_collision(self.x + 80)

suggested\_state\_current\_row\_Left2 = current\_row.check\_collision(self.x -80)

# As the rabbit can think a row is clear and move forwards when a car is still too close, extra variables are used

# to check most of the space in front of the bunny is safe to move to

suggested\_state\_next\_row\_side\_left = next\_row.check\_collision(self.x - 40)

suggested\_state\_next\_row\_side\_right = next\_row.check\_collision(self.x + 40)

suggested\_state\_next\_row\_side\_left2 = next\_row.check\_collision(self.x - 80)

suggested\_state\_next\_row\_side\_right2 = next\_row.check\_collision(self.x + 80)

# Used if the AI needs to go backwards

suggested\_state\_previous\_row = next\_row.check\_collision(self.y + 40)

# In an attempt to make the AI more human and not input too quickly, the following variable is generated every loop

# and will only let the action run if 1 has been picked

waitInt = randint (1, 3)

# Used for the hedge section

pickDirection = randint (1, 2)

moveRandomAmount = randint (1, 6)

# Check if row type is of type...

if rowType == "Grass" or rowType == "Dirt" or rowType == "Pavement":

# WAIT

# Check the row in front is clear...

if (str(suggested\_state\_next\_row).find("ALIVE") == -1):

# The following lines were designed to stop the AI running into a car in front too early by checking the

# left and right lines in front before moving

# This code made the AI more effective in the cars section but was causing the "Water" state not to work

#(str(suggested\_state\_next\_row\_side\_right).find("ALIVE") == -1) or \

#(str(suggested\_state\_next\_row\_side\_right2).find("ALIVE") == -1 or \

#(str(suggested\_state\_next\_row\_side\_left2).find("ALIVE") == -1)) or \

#(str(suggested\_state\_next\_row).find("ALIVE") == -1):

print("Danger ahead!")

# Reset the hedge flag

self.iteration = -1

# CHECK UP

elif waitInt == 1:

print("Safe to move!")

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

print("UP")

# This flag is used to determin how many times the AI has attempted to use this state

self.iteration += 1

# The following code was made as an attempt to move the AI around hedges

# The self.iteration is checked and will run if the rabbit has been stuck for 10 turns

# Each statement will check which side of the screen the rabbit is closest to

if self.iteration >= 10 and self.x > WIDTH / 2:

print("There's something in my way!")

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

print("LEFT")

# Reset the hedge flag

self.iteration = -1

if self.iteration >= 10 and self.x < WIDTH / 2:

print("There's something in my way!")

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

print("RIGHT")

self.iteration = -1

# The following code was used as an attempt to make the AI not get stuck directly in the centre of the screen

# If not used, the rabbit will be stuck in an infinate loop of moving left and right in the centre

if self.iteration >= 10 and self.x == WIDTH / 2 and pickDirection == 1:

# Run this code x number of times

for x in range(0, 6):

print("There's something in my way!")

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

print("RIGHT")

# Reset the hedge flag

self.iteration = -1

if self.iteration >= 10 and self.x == WIDTH / 2 and pickDirection == 2:

for x in range(0, 6):

print("There's something in my way!")

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

print("LEFT")

# Reset the hedge flag

self.iteration = -1

# Check if row type is of type...

if rowType == "Water":

# CHECK UP

if (str(suggested\_state\_next\_row).find("ALIVE") == -1):

print("Danger ahead!")

elif waitInt == 1:

print("Safe to move!")

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

print("UP")

if rowType == "Road" or rowType == "Rail":

# The following code was made as an attempt to make the AI double-back if

# it was stuck between a car at it's side and a car in front of it

# This code was removed in the end as the AI would use it too often and wouldn't make enough progress

# Uncomment to see in action

# SANDWHICHED BETWEEN 2 CARS (LEFT AND TOP)

## if (str(suggested\_state\_current\_row\_Left).find("SPLAT") == -1) and \

## (str(suggested\_state\_next\_row).find("SPLAT") == -1) and \

## (str(suggested\_state\_previous\_row).find("SPLAT") == -1):

## pass

##

## elif waitInt == 1:

## self.input\_queue.append(DIRECTION\_DOWN)

## print("DOWN")

## print("I'm surrounded, Better go back!")

## self.iteration = -1

# SANDWHICHED BETWEEN 2 CARS (RIGHT AND TOP)

## if (str(suggested\_state\_current\_row\_Right).find("SPLAT") == -1) and \

## (str(suggested\_state\_next\_row).find("SPLAT") == -1) and \

## (str(suggested\_state\_previous\_row).find("SPLAT") == -1):

## pass

##

## elif waitInt == 1:

## self.input\_queue.append(DIRECTION\_DOWN)

## print("DOWN")

## print("I'm surrounded, Better go back!")

## self.iteration = -1

# CHECK LEFT

if (str(suggested\_state\_current\_row\_Left).find("SPLAT") == -1) or \

(str(suggested\_state\_current\_row\_Left2).find("SPLAT") == -1):

# No conditions needed for this outcome

pass

# Move away from oncoming traffic

elif waitInt == 1:

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

print("RIGHT")

print("Danger to my left!")

# CHECK RIGHT

if (str(suggested\_state\_current\_row\_Right).find("SPLAT") == -1) or \

(str(suggested\_state\_current\_row\_Right2).find("SPLAT") == -1):

# No conditions needed for this outcome

pass

# Move away from oncoming traffic

elif waitInt == 1:

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

print("LEFT")

print("Danger to my right!")

# CHECK UP

if (str(suggested\_state\_next\_row).find("ALIVE") == -1) or \

(str(suggested\_state\_next\_row\_side\_left).find("ALIVE") == -1) or \

(str(suggested\_state\_next\_row\_side\_right).find("ALIVE") == -1) or \

(str(suggested\_state\_next\_row\_side\_right2).find("ALIVE") == -1 or \

(str(suggested\_state\_next\_row\_side\_left2).find("ALIVE") == -1)):

print("Danger ahead!")

elif waitInt == 1:

print("Safe to move!")

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

print("UP")

# References

Ager, R. (Director). (2017). *Androids and Artifical Intelligance: A Modern Myth* [Motion Picture]. Retrieved from https://gumroad.com/l/SNoq

AI and Games. (2019, May 29th). *The AI of Half-Life: Finite State Machines | AI 101*. Retrieved from YouTube: https://youtu.be/JyF0oyarz4U

Belwariar, R. (2018, September 7th). *A\* Search Algorithm*. Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/a-search-algorithm/

Buchanan, B. G. (2006). A (Very) Brief History of Artificial Intelligence. *AI Magazine Volume 26 Number 4*, 8. Retrieved from https://www.aaai.org/ojs/index.php/aimagazine/article/view/1848

Computer Science. (2017, January 6th). *Graph Data Structure 6. The A\* Pathfinding Algorithm*. Retrieved from YouTube: https://youtu.be/eSOJ3ARN5FM

Crookes, D., Gillett, A., & Upton, L. (2019). *Code the Classics Volume 1.* (P. King, Ed.) Cambridge: Rasberry Pi Press. Retrieved from https://wireframe.raspberrypi.org/books/code-the-classics1

Dharmkar, R. (2017, December 08). *How to check if a string only contains certain characters in Python?* Retrieved from tutorialspoint.com: https://www.tutorialspoint.com/How-to-check-if-a-string-only-contains-certain-characters-in-Python

Good, O. S. (2019, September 21st). *All the ways the AI cheated on you in Street Fighter 2*. Retrieved from Polygon: https://www.polygon.com/2019/9/21/20877210/street-fighter-2-ai-cheats-arcade-snes

Santatan, M. (2020). *Theory of Computation: Finite State Machines*. Retrieved from Stack Abuse: https://stackabuse.com/theory-of-computation-finite-state-machines/

sf2platinum. (2017, January 20th). *Street Fighter II: The AI Engine*. Retrieved from Wordpress: https://sf2platinum.wordpress.com/2017/01/20/the-ai-engine/

Thompson, T., & Scales, D. (2014, August 26th). SpelunkyBots API - An AI Toolset for Spelunky. *2014 IEEE Conference on Computational Intelligence and Games*, 8. Retrieved from https://ieeexplore.ieee.org/abstract/document/6932872

Wood, A. (2018, October 15th). *Red Dead Redemption 2 has a 60-hour story, 500,000 lines of dialogue* . Retrieved from gamesradar.com: https://www.gamesradar.com/red-dead-redemption-2-has-a-60-hour-story-500000-lines-of-dialogue/