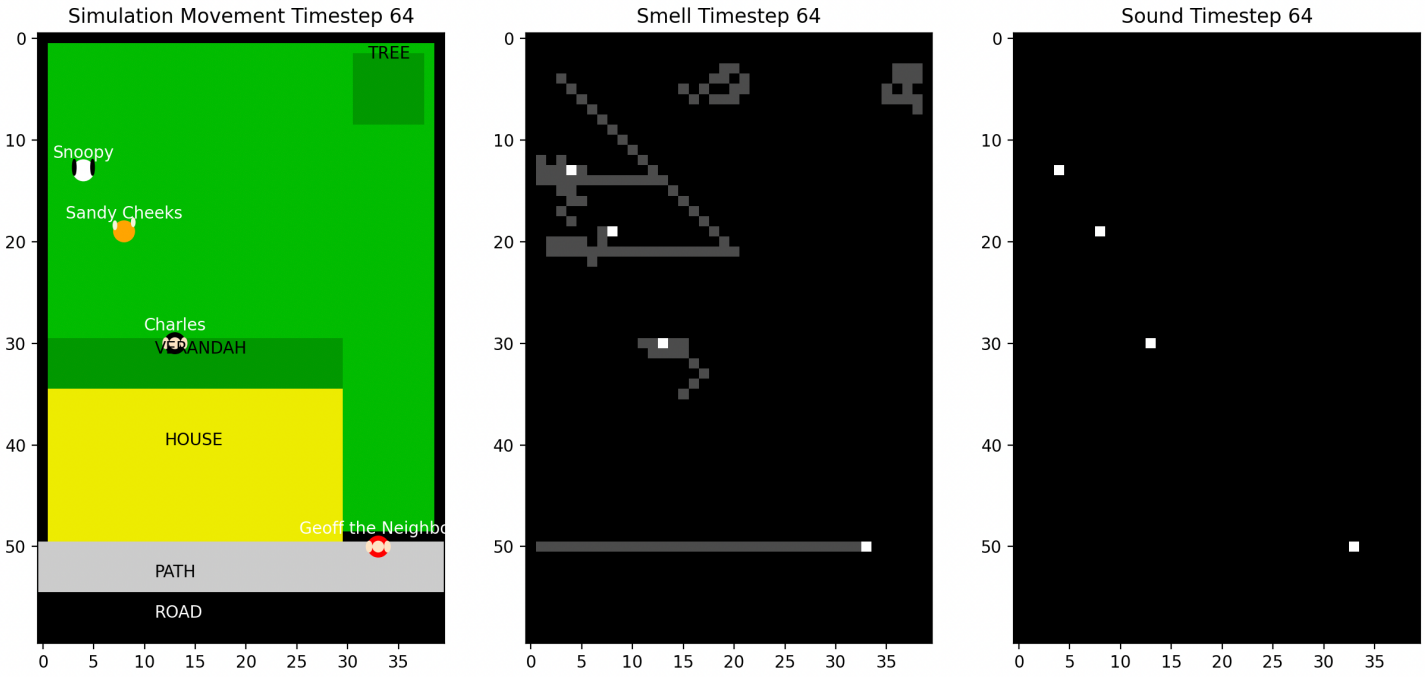
Fundamentals of Programming Project Report

Puppy and Squirrel Simulation



Muhammad Annas Atif
22224125
5th May 2024

Table of Contents

Overview

The fundamental purpose of this project is to combine all gathered knowledge developed over the course of this unit and cohesively combine it into an interesting and fun, application-based project to assess our understanding of the learnt concepts.

There are a series of features implemented into the program that increase the complexity and realism of the simulation.

The puppy, squirrel, owner, and neighbor are all themselves features within the simulation. They have their own movement functions that allow them to move in varying ways i.e., the puppy can move in direction north, east, south, and west and all diagonals while the neighbor can only move in the east & west directions and the human and squirrel can only move in the north, east, west, and south directions. (Von Neumann and Moore Moves) (COMP1005b, 2024). Furthermore, each of them has their own physical appearances and names which are user adjustable within the command line.

There are also implementations of colliders that prevent them from moving off the simulation space and ones that prevent some elements from entering/exiting certain regions on the plot. This can be seen with the following examples:

- The squirrel is the only one allowed to be in the zone for the tree.
- The dog and squirrel can enter the verandah but not enter inside the house.
- Only the human is allowed inside the house.
- Only the neighbor is allowed on the path.

The owner specifically has a feature that allows for the user to adjust how quickly they move towards the verandah on the map to place the treat and, acorn. The treat and acorn will only spawn once the owner reaches the edge of the verandah on the map.

The neighbor is the only individual who is allowed to throw a toy over the fence at a random timestep within the simulation parameters.

Once the treat, toy and acorn have been spawned on the map, the puppy will move towards the treat and consume is increasing its health, and the squirrel will move towards the acorn to consume it increasing its health.

An important feature of the treat, acorn and toy is that they are all visually different and they also randomly spawn within region on the map accessible by the squirrel and puppy. The puppy then will go towards the toy and interact with it for a user defined number of timesteps and will then drop it or bury it.

There is also a command line before starting the simulation allowing for the user to either choose a simulation based upon predetermined values assigned to user definable parameters or the user may choose to edit the simulation parameters.

The code for the assignment initially was based off the base code package (COMP1005a, 2024), however eventually throughout the course of the assignment, it has almost entirely been replaced with rewritten, custom code that is more efficient.

User Guide

Pre-Requisites:

**Note all requirements were based upon testing and may not work correctly if they do not match the following system requirements.*

- System: macOS (ARM, x86, x64), Windows (x86, x64), Linux (Tested on Curtin VM's)
- Python: v3.x or later
- Matplotlib: v3.7.3 or later
- Numpy: v1.26 or later
- Default python libraries: python v3.x or later.

Recommended Software:

- Visual Studio Code – 1.88.1 or later.

** It is advised that recommended software be used to run the simulation otherwise issues may arise. Also note that older versions of recommended software **may** work, however they have been **untested** and your milage may vary.*

Running Instructions

Before running the code, please ensure that all pre-requisites are met. For modules such as *numpy* and *matplotlib*, as well as required software such as *Python*, installation instructions for your specific operating system may be found online with a simple Google Search.

1. Extract the zip file named 22224125_FinalAssignment.zip.
2. Locate the file named "Snoopy" and open it.
3. Find the file named "snoo.py".
4. If a Windows computer is being used, simply right clicking, and pressing "Run with Python" should run the program.

   If on a Mac open the Python IDLE app, open the file "Snoo.py" by pressing "File" in the menu bar and pressing "Open". Then Press either F5 or press Run in the menu bar and press "Run

   If on Linux, open terminal, using the command "cd" open the folder named "Snoopy" which contains the code. Then press enter. Once done type "python3 snoo.py" and press enter. E.g. *cd Downloads/Simulation/Snoopy/* followed by *python3 snoo.py.*

   Note this method also is applicable for macOS

   If this not an option, install the recommended software. When setting up Visual Studio Code, an installation manual can be found online that specify detailed settings. However, for this simulation, no additional tinkering is required other that installing the Python module by searching for "Python" within the "Extensions Tab"

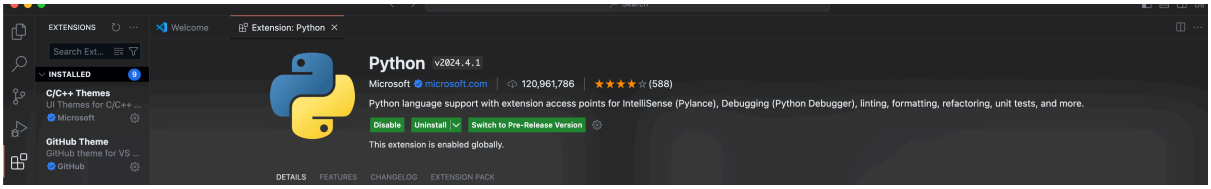


Fig 1.0 – Visual Studio Code Extensions Tab with Python

5. To run the code in this software, press the "Explorer Tab" which is the first icon located on the vertical stack and press "Open Folder".
6. Locate the folder named "Snoopy" that was extracted and press open.
7. In the explorer tile, press "snoo.py" and find the play button located on the toolbar.

Fig 1.1 – File Name in Explorer Tab and Running Procedure

8. A terminal window should pop-up in the bottom of the window. You may resize the window by hovering over the edge and dragging the tile up.

** See Fig 1.2 – Example Terminal*

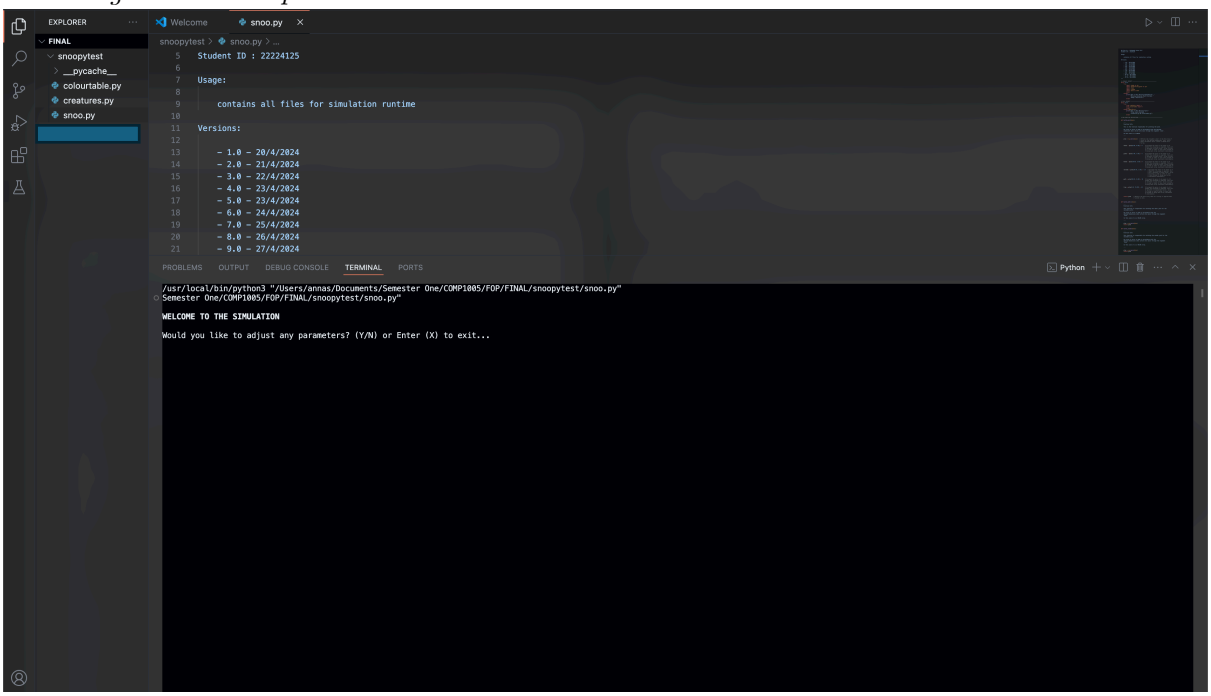


Fig 1.2 – Example Terminal

9. The simulation program is now running.
10. Follow the instructions presented in the first input.

11. By selecting "N" the simulation will run with default parameters.

** See Table 1.0 – Default Parameters for a comprehensive breakdown of the default parameters as well as the explanation of the parameters if "Y" is selected in the terminal.*

Table 1.0 – Default Parameters

Parameter	Explanation	Value
Puppy	Puppy object	Name: Snoopy Colour: white/black Position: (5,15) Previous Position: (5,14) Age: 74
Squirrel	Squirrel Object	Name: Sandy Cheeks Colour: orange/beige Position: (5,35) Previous Position: (5,34) Age: 43
Human	Human Object	Name: Charles Colour: black/bisque Position: (35,15) Previous Position: (35,15) Age: 30
Neighbor	Neighbor Object	Name: Geoff Colour: red/bisque Position: (50,1) Previous Position: (50,0) Age: 35
Size	Plot dimensions	(60,40)
Treatno	Number of treats	1
Acornno	Number of acorns	1
Toyno	Number of toys	1
Simrun	Number of simulations timesteps. i.e. how many times it will run.	200
Timesleep	How long each timestep will last (seconds)	0
Iteration	At what n'th timestep will the human/neighbor move. i.e. every second timestep = 2.	2
Holdtime	For how many timesteps will the puppy play with the toy.	30

12. By selecting "Y". Please read the instructions of the "Customisation Studio" before proceeding.

Fig 1.3 – Customisation Studio with General Parameter Settings Example

13. General parameters will show up first, please ensure numerical integer inputs are given otherwise an error will ask you to re-enter the correct value.
14. Please read the instructions given for each parameter input before entering a value. Default values are also provided in the instructions.
15. The "Puppy Creator" will then launch.
16. Enter the name of the puppy that will be in the custom simulation.

Fig 1.4 – Puppy Input for Name

17. The next input will ask you to enter "Y" if you would list to see a list of accepted colours for the colour(s) of the puppy (Matplotlib n.d.). Otherwise press any character and press enter to continue
18. A pop-up will show up and then press the "X" on the window (if on Windows and Linux) and the red dot if on MacOS once the desired colours are decided.

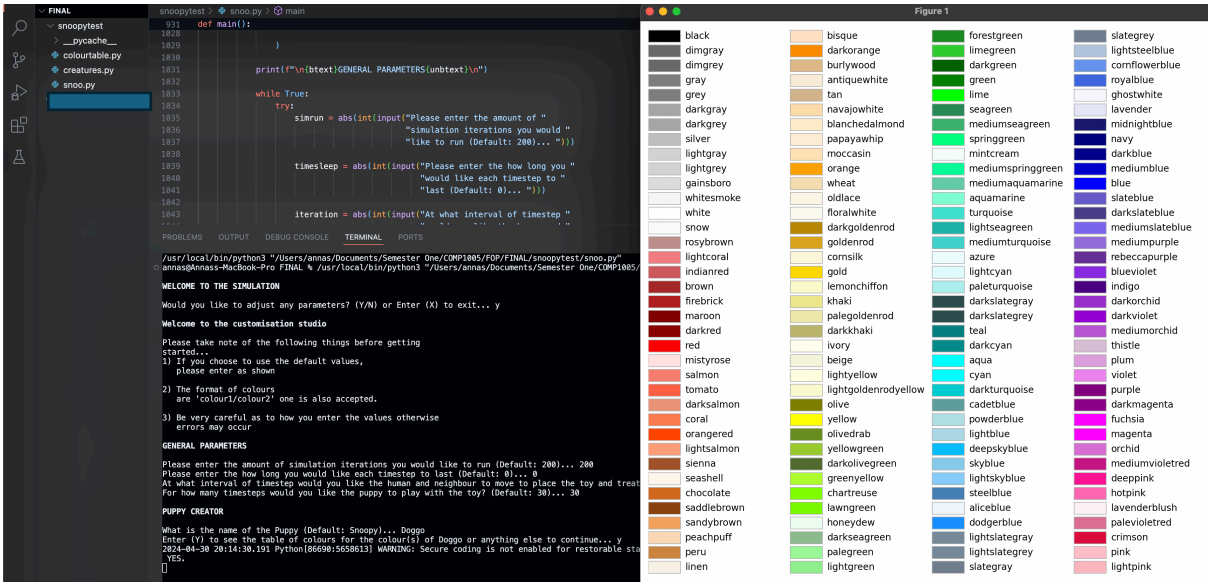


Fig 1.5 – Table of Acceptable CSS4 Values (Matplotlib n.d.) with Prior Terminal Inputs.

19. Enter the colours in the next input making sure to ensure the format is as stated and in the examples.

Fig 1.6 – Example Input for Colours

20. Enter the age of the puppy making sure to keep the value a positive integer value.

Fig 1.7 – Example for Age Input

21. Follow Step 15 – 20 for the "Squirrel Creator", "Human Creator" and "Neighbor Creator"

22. Once the simulation has started, the terminal window (*as seen in Fig 1.8 – Example Terminal Output)* will show the following info:

- Current X and Y position of the Puppy, Squirrel, Human and Neighbor
- The valid moves of the Puppy, Squirrel, Human and Neighbor
- The move made by the Puppy, Squirrel, Human and Neighbor
- The Current Health of the Puppy and Squirrel

Fig 1.8 – Example Terminal Output

23. The plot will also start showing the simulation. There will be 3 plots. One for the movement, one for smell and one for sound.

* *See Fig 1.9 – Example Simulation Plot*

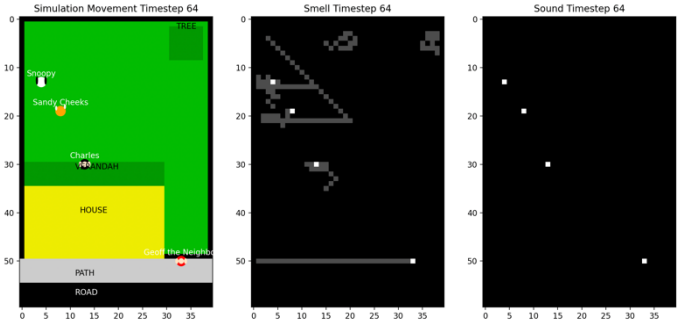


Fig 1.9 – Example Simulation Plot

Traceability Matrix

The traceability matrix gives an overview of the implemented and planned code features as well as the testing status of code and how it was/will be tested.

Features of the Traceability Matrix

- Feature Name – The name of the relevant feature
- S – Status | PS = Pass | F = Fail | PA = Pass (Partial) | SK = Skipped | N = N/A
- File Location – The file in which the code is located as well as the function name.
- Testing Ref – How/Where was the file tested.
- C Date – When was the feature marked as a PS/PA i.e., when was it completed.

Table 1.1 - Traceability Matrix

Feature Name		S	File Location	Testing Ref	C Date
1.0)	**Puppy**	PS	creatures.py – Class Puppy	///	29/4/24
1.1)	Plotting Position, Smell and Sound	PS	creatures.py – Class Puppy - Puppy.plot_me() snoo.py - creature_plot() (Whole Function) - listname[c].plot_me(axis,sizing)	Running of _main_ to observe output.	20/4/24
1.2)	Styling	PS	creatures.py – Class Puppy - Puppy.plot_me() - paxis.add_patch() → (all) - Puppy._init_() → lines 87 - 95	Running of _main_ to observe output by appending a test puppy to the creatures list.	20/4/24
1.3)	Movement	PS	creatures.py – Class Puppy - Puppy.step_change() → (whole function) - Puppy._init_() lines 99 – 106 snoo.py – creature_move()	Running of _main_ to observe output by appending a test puppy to the creatures list.	21/4/24
1.4)	Colliders	PS	creatures.py – Class Puppy - Puppy.step_change() → lines 126 - 154	Running of _main_ to observe output b checking boundary collisions by appending a test puppy to the creatures list.	22/4/24
1.5)	Health/Age	PS	creatures.py – Class Puppy - Puppy._init_() → Line 97 and 107 - health() (Whole function) - healthloss() (Whole function)	Running of _main_ to observe output in the terminal by appending a test puppy to the creatures list.	28/4/24

			snoo.py – creaturehealth() (whole function)		
1.6)	Treats	PS	snoo.py - main() - treatno = 1 - treat_plot() (Whole Function) - plot_treat_acorn_toy( ) → Line 644 - clear_treats() (Whole Function)	Running of _main_ to observe output by appending a test puppy to the creatures list and waiting for it to walk over a treat.	26/4/24
1.7)	Toys	PS	snoo.py - main() – toyno = 1 - toy_plot() (Whole Function) - plot_toy() (Whole Function) - clear_toys() (Whole Function)	Running of _main_ to observe output by appending a test puppy to the creatures list and waiting for it to walk over a toy.	26/4/24
1.8)	Movement to Treat	PS	snoo.py - plot_treat_acorn_toy( ) → Lines 647-651 creatures.py → Puppy Class - step_change() → Lines 116 – 118 - move_towards_treat() (Whole Function) - _init_() Lines → 100, 102	Running of _main_ to observe output by appending a test puppy to the creatures list and through trial and error, waited for it to move once the treat had been plotted.	26/4/24
1.9)	Consumption of Treat	PS	creatures.py → Puppy Class - _init_() Lines → 105 - treat_collected() (Whole Function) snoo.py - clear_treats() (Whole Function)	Running of _main_ to observe output by appending a test puppy to the creatures list and through trial and error, waited for it to have its position equal to the treat.	25/4/24
1.10)	Movement to Toy	PS	snoo.py - plot_toy() → Lines 684 – 689 creatures.py → Puppy Class - step_change() → Lines 120 – 122 - move_towards_toys() (Whole Function) - _init_() → Lines 101, 103	Running of _main_ to observe output by appending a test puppy to the creatures list and through trial and error, waited for it to move once the toy had been plotted.	26/4/24
1.11)	Toy Interaction	PS	creatures.py → Puppy Class - _init_() → Lines 104 - toy_collected() (Whole Function)	Running of _main_ to observe output by appending a test puppy to the creatures list and through trial and error, waited for it to move once the toy	26/4/24

			snoo.py - plot_placed_toy() (Whole Function) - place_toy() (Whole Function) - clear_toys() (Whole Function)	had been plotted. Furthermore, observations were also taken as to whether when the puppy dropped the toy, the toy had been buried or placed.	
1.12)	Sleep	PS	creatures.py → Puppy Class - _init_() → Lines 106 - Step_change() → Lines 124, 125 snoo.py - creaturehealth() → Lines 769 – 770 -	Running of _main_ to observe whether the movement of the puppy stops after the health drops below 50.	29/4/24
2.0)	**Squirrel**	PS	Creatures.py – Class Squirrel		29/4/24
2.1)	Plotting Position, Smell and Sound	PS	creatures.py – Class Squirrel - Squirrel.plot_me() snoo.py - human_plot() (Whole Function) - human[c].plot_me(axis, sizing)	Running of _main_ to observe output.	20/4/24
2.2)	Styling	PS	creatures.py – Class Squirrel - Squirrel.plot_me() - paxis.add_patch() → (all) - Squirrel._init_() → lines 293 – 300	Running of _main_ to observe output by appending a test squirrel to the creatures list.	20/4/24
2.3)	Movement	PS	creatures.py – Class Squirrel - Squirrel.step_change() → (whole function) - Squirrel._init_() lines 301, 303 - 307 snoo.py – creature_move()	Running of _main_ to observe output by appending a test squirrel to the creatures list.	21/4/24
2.4)	Colliders	PS	creatures.py – Class Squirrel - step_change() lines → 330 – 352	Running of _main_ to observe output by waiting for boundary collisions . Done by appending a test squirrel to the creatures list.	22/4/24
2.5)	Health/Age	PS	creatures.py – Class Squirrel - Squirrel._init_() → Line 301 and 307 - health() (Whole function)	Running of _main_ to observe output in the terminal by appending a test squirrel to the creatures list.	28/4/24

			- healthloss() (Whole function) snoo.py – creaturehealth() (whole function)		
2.6)	Acorns	PS	snoo.py - main() - acornno = 1 - acorn_plot() (Whole Function) - plot_treat_acorn_toy( ) → Line 645 - clear_acorns() (Whole Function)	Running of _main_ to observe output by appending a test squirrel to the creatures list and waiting for it to walk over an acorn.	26/4/24
2.7)	Movement to Acorn	PS	snoo.py - plot_treat_acorn_toy( ) → Lines 653-657 creatures.py → Squirrel Class - step_change() → Lines 320 – 323 - move_towards_acorns( ) (Whole Function) - _init_() Lines → 303, 305	Running of _main_ to observe output by appending a test squirrel to the creatures list and through trial and error, waited for it to move once the acorn had been plotted.	26/4/24
2.8)	Consumption of Acorn	PS	creatures.py → Squirrel Class - _init_() Lines → 306 - treat_collected() (Whole Function) snoo.py - clear_acorns() (Whole Function)	Running of _main_ to observe output by appending a test squirrel to the creatures list and through trial and error, waited for it to have its position equal to the acorn.	25/4/24
2.9)	Sleep	PS	creatures.py → Squirrel Class - _init_ () → Lines 307 - step_change() → Lines 326, 327 snoo.py - creaturehealth() → Lines 769 – 770	Running of _main_ to observe whether the movement of the puppy stops after the health drops below 50.	29/4/24
2.10)	Movement Away from Puppy	PA	creatures.py → Squirrel Class - step_change() → Lines 354 – 376	Running of _main_ to observe whether the squirrel comes into a 3-unit radius of the puppy and then does not get any closer. This function has *sometimes* works for some reason. Hence, partial pass given.	29/4/24
3.0)	**Human**	PS	Creatures.py – Class Human	///	26/4/24
3.1)	Plotting Position, Smell and Sound	PS	creatures.py – Class Human - Human.plot_me()	Running of _main_ to observe output.	20/4/24

			snoo.py - human_plot() (Whole Function) - humanlist[c].plot_me(axis,sizing)		
3.2)	Styling	PS	creatures.py – Class Human - Human.plot_me() - paxis.add_patch() → (all) - Human._init_() → lines 468 - 475	Running of _main_ to observe output by appending a test human to the human list.	20/4/24
3.3)	Movement	PS	creatures.py – Class Human - step_change_normal() (Whole Function) snoo.py - human_move_normal() (Whole Function)	Running of _main_ to observe output by appending a test human to the human list and monitoring	26/4/24
3.4)	Colliders	PS	creatures.py – Class Human - step_change() lines → 500 – 514	Running of _main_ to observe output by waiting for boundary collisions . Done by appending a test human to the human list.	22/4/24
3.5)	Placement of Acorn and Treat + Movements	PS	creatures.py – Class Human - step_change_drop_treat() (Whole Function) snoo.py - simulation() Lines → 881 – 890 - human_move_drop_treat() (Whole Function)	Running of _main_ to observe the human going to place the treat and acorn when the test human reaches the edge of the verandah.	25/4/24
4.0)	**Neighbor**	PS	Creatures.py – Class Neighbor	///	28/4/24
4.1)	Plotting Position, Smell and Sound	PS	creatures.py – Class Neighbor - Neighbor.plot_me() snoo.py - neighbor_plot() (Whole Function) - neighborlist[c].plot_me(axis,sizing)	Running of _main_ to observe output.	27/4/24
4.2)	Styling	PS	creatures.py – Class Neighbor - Neighbor.plot_me() - paxis.add_patch() → (all) - Human._init_() → lines 579 - 586	Running of _main_ to observe output by appending a test neighbor to the neighbor list.	27/4/24
4.3)	Movement	PS	creatures.py – Class Neighbor - step_change_right() (Whole Function) snoo.py - human_move() (Whole Function)	Running of __main__ to observe output by appending a test neighbor to the neighbor list and monitoring movement.	27/4/24

	4.4) Colliders	PS	creatures.py – Class Human - step_change() lines → 603 – 607	Running of _main_ to observe output by waiting for boundary collisions . Done by appending a test neighbor to the neighbor list.	27/4/24
	4.5) Placement of Toy	PS	snoo.py - plot_toy() (Whole Function) - simulation() Lines → 892 - 894	Running of _main_ to observe whether the toy had been placed in the position assigned by the simulation lines (x axis)	28/4/24
5.0)	**Plot**	PS	Snoo.py – gen_yard() function	///	24/4/24
	5.1) House	PS	Snoo.py – - gen_yard() Line →75	Running of _main_ to observe the plot produced.	20/4/24
	5.2) Verandah	PS	Snoo.py – - gen_yard() Line →93	Running of _main_ to observe the plot produced.	20/4/24
	5.3) Fences	PS	Snoo.py – - gen_yard() Line →75	Running of _main_ to observe the plot produced.	20/4/24
	5.4) Footpath	PS	Snoo.py – - gen_yard() Line →100	Running of _main_ to observe the plot produced.	20/4/24
	5.5) Road	PS	Snoo.py – - gen_yard() Line →75	Running of _main_ to observe the plot produced.	20/4/24
	5.6) Tree	PS	Snoo.py – - gen_yard() Line →106	Running of _main_ to observe the plot produced.	20/4/24
	5.7) Smells	PS	snoo.py - gen_smells() (Whole Function) - update_smells() (Whole function) - creature_plot() → Line 419 - human_plot() → Line 444 - neighbor_plot() → Line 472 - plot_treat_acorn_toy( ) → Line 649, 655, 686	Running of _main_ to observe the smells plot produced.	24/4/24
	5.8) Sounds	PS	snoo.py - gen_sounds() (Whole Function) - update_sounds() (Whole Function) - creature_plot() → Line 420 - human_plot() → Line 445 - neighbor_plot() → Line 473 - plot_treat_acorn_toy( ) → Line 649, 655 - plot_toy() → Line 687	Running of _main_ to observe the sounds plot produced.	24/4/24

6.0)	**Command Line / General**	PS	snoo.py **Note to ensure the perpetual running of the simulation unless the user inputs the exit commands, the terminal incorporates the use of a while loop. This is a valid use case as otherwise, the use of "if" statements and other supplementary functions would result in less robust, less readable, and more complicated code with more points at which there could be technical issues. The use of the while loop results in a seamless and simplistic approach to these potential flaws.* *Furthermore, for exception handling while loops and breaks have been used in a valid manner to allow for checks to only occur at the launch of the program allowing for more efficient code rather than having checks being performed at an unnecessary amount of time intervals.*	///	30/4/24
	6.1) Exception Handling Files and Libraries	PS	Line(s) → 27 – 40 and 43 - 52 **Note breaks have been used here. This is a valid use case as the break allows for the exception handling to run.*	Tested by renaming a file in the directory to check for handling of errors in the terminal.	30/4/24
	6.2) Exception Handling Inputs	PS	Line(s) → 1033 – 1058 **Note breaks have been used here. This is a valid use case as the break allows for the exception handling to run.*	Tested by entering non-integer and negative values into the terminal window prompts to check for error handling.	28/4/24
	6.3) Simulation w/ Predefined Values	PS	Line(s) → 1197 - 1230	Tested by checking terminal outputs and plot behavior and compared it to the predefined values.	28/4/24
	6.4) Customise Simulation Creatures/ Humans/ Neighbor	PS	Line(s) → 1059 -1174	Tested by checking the output of the yard to see whether relevant changes show effect. i.e., checking the colour of creatures etc.	28/4/24
	6.5) Customise General Parameters (timestep etc.)	PS	Line(s) → 1035 - 1049	Tested by checking terminal outputs and plot outputs to see whether they corresponded to the relevant parameter changes.	28/4/24

Discussion

Generally speaking, there is a fairly large number of features implemented into this simulation model ranging from basic features such as adjusting how long the plot will remain on the screen to having a movement algorithm in which the puppy will move towards treats and toys. In this discussion, the various features implemented will be discussed in detail looking at their purpose in the simulation, logic behind them and how they were seamlessly integrated into the simulation. Finally, a UML class diagram will be presented showcasing the objects in the simulation and their relations.

Features

This section will focus deeply on the feature and its purpose, its logic as well as how they were integrated.

(1.0) Puppy and Associated Features

The puppy is a major feature within the simulation, and it is also a requirement of the brief. The puppy is a visible object within the plot and serves to interact with elements on the plot using its senses which are: sight, sound, smell, and touch.

(1.1) The plotting, position, smell, and sound (senses) of the puppy were integrated within the class object inside of "creatures.py" . The plotting specifically was done within the "plot_me()".

Within *snoo.py* the *creature_plot()* function would be called to plot the creature onto the correct set of axes with sound and smell as well.

(1.2) The styling of the puppy was again done within *creatures.py* under the puppy class object. Within the *plot_me()* function, the *patches* module of *Matplotlib* was used in order to create "*plottable*" shapes on the graph.

The logic behind this was to avoid creating several complicated arrays that could theoretically serve the same purpose. It was also done to allow higher quality shapes and curves.

(1.3) The normal movement algorithm of the puppy is quite a robust one. The main component of the movement algorithm is located within the puppy class object in *creatures.py*.

Within the movement algorithm lies a list of valid moves which contain coordinates movements. a random move is selected from the list and the coordinate is sliced using slicing. The relevant x and y coordinates are then added on to the *self.pos* variable.

The puppy differs compared to other creatures/humans/neighbors in the sense that diagonal movements (Moore Movement) are allowed alongside the typical north, east, south, and west movements (Von Neumann). (COMP1005b, 2024)

Finally, this movement algorithm is translated into a viewable state in *snoo.py* where the function *creature_move()* is called.

(1.4) The movement colliders for the puppy are quite simplistic whilst being sufficiently robust and easy to modify. The collider system is located on lines 126 – 154 of the *step_change()* function in *creatures.py* under the puppy class object.

By using the *self.pos* variable, the respective x and y coordinates of the puppy can be sliced out and using if statements the list of valid moves can be constructed to prevent the puppy from moving into a location it should not be. i.e., inside the house, outside the fence, and outside the plot. Where any collider conditions are not met, the code allows for all possible moves.

(1.5) The health and age of the puppy is implemented in *creatures.py* under the puppy class object. On line 97, the age variable is assigned to the value of *self.age* and on line 107 the health of the puppy is set to 100 under the variable *self.health*.

The function *health()* is used to return an integer value of *self.health*. The *currenthealth* function calculates the health at the given timestep. Within *snoo.py* the function *creaturehealth()* calls the *healthloss* function for each creature in the list, which means that the puppy's health is then calculated at each timestep.

(1.6) The puppy also has the ability to interact with treats that spawn on the map and consume them for additional health. Within *snoo.py* under the *_main_*() function, the number of treats is defined to be 1.

The *treat_plot()* function assigns the position of the to-be-spawned treat to a random position in an accessible area by the puppy.

In the *plot_treat_acorn_toy()* function, on lines 644, the list from *treat_plot()* is generated. A value of 2 set to the coordinate position of the treat in the array for the yard, this shows a blue colour on the main yard plot, the value of the smell is set to 10 for the coordinate position of the treat which is reflected on the smells plot as white.

Finally, on lines 650 and 651, the puppy's movement system is altered to the alternative movement algorithm that allows it to move towards the treat

Finally, once the position of the treat and puppy is the same, it is cleared off the map by the function *clear_treats()*

(1.7) Toys are another thing the puppy can interact with on the map once they have spawned. Within *snoo.py* and within the *_main_()* function, the number of toys is defined to be 1.

The *toy_plot()* function serves the purpose to assign positions of the toys.

Subsequently in the function *plot_toy()*, on line 681, the number of toys is printed and on line 682, the *toy_plot()* function is called to generate the list of toy positions. The yard, smell and sound array have the position of the toy assigned. These are then all respectively displayed on the respective plots.

The puppy's movement system is altered to the alternative movement algorithm that allows it to move towards the desired toy. Once the puppy reaches the toy the function *clear_toys()* clear them off the plots.

(1.8) The movement towards the treat is done through an adjustment to the movement algorithm that overrides the normal movement and allows the puppy to directly go towards the treat.

Within *snoo.py*, once the treat has been plotted and *listname[0].move_towards_treat = True*, within *creatures.py* under the puppy class object in the function *step_change()*, the *move_towards_treats()* movement algorithm is enabled as the condition for the if statement on line 116 is met.

(1.9) The consumption of the treat is done in a robust yet simple way. In *creatures.py* on line 105, when the position of the treat and puppy are equal, then the Boolean value for *self.treat_collected* is set to *True*. This is done in the *move_towards_treat()* function when the values of *dx* and *dy* are equal to 0, and then returned by the function *treat_collected()*. This as a result runs the *clear_treats()*

(1.10) The movement towards the toy is done through an adjustment to the movement algorithm that overrides the normal movement and allows the puppy to directly go towards the toy.

Within *snoo.py*, once the toy has been plotted and *listname[0].move_towards_toy = True*, within *creatures.py* under the puppy class object in the function *step_change()*, the *move_towards_toys()* movement algorithm is enabled as the condition for the if statement on line 120 is met.

(1.11) The puppy also has the ability to interact with the toy on the plot once it has been thrown over the fence by the neighbor. The addition of the toy increases the overall realism of the simulation.

Within *creatures.py*, and the puppy class object, and in the *_init_()* function, the initial Boolean value for *self.toy_collected* is set to *False*, this allows the puppy to move towards the toy. Once the position of the toy and that of the puppy is equal, i.e., *dy* and *dx* are equal to 0, the Boolean value is set to *True*. This value is returned by the *toy_collected()* function.

In *snoo.py* from lines 902 – 904, the counter which is implemented on line 868 is set to the current number of timesteps for which the puppy has held the treat for. On

line 906 if the counter equals the user defined value for how long the puppy should play with the treat for *(holdtime)*, then the *plot_placed_toy()* function is called. The *plot_placed_toy()* function is responsible for reflecting the position of the toy when the puppy is done holding it for the desired amount of timesteps. The puppy has the choice of either burying it or simple placing it down at its position.

(1.12) When the health of the puppy falls below 50, the puppy will stop moving and essentially go to sleep. In *creatures.py* on line 106, the Boolean value of *self.tired* is set as default to *False*.

In *snoo.py* under the *creaturehealth()* function, if the value of the health falls to a value < 50 on lines 769 and 770, the Boolean value of *self.tired* is set to *True*.

Back in *creatures.py* this is checked in the *step_change()* function on lines 124 and 125 where if the Boolean value is *True*, then the *sitdown()* function is active (line 205-207), this results in the position of the puppy being fixed to its position when the value is < 50 for the health not allowing it to move.

(2.0) Squirrel and Associated Features

The squirrel is a major feature within the simulation, and it is also a requirement of the brief. The squirrel is a visible object within the plot and serves to interact with elements on the plot using its senses which are: sight, sound, smell, and touch.

(2.1) The plotting, position, smell, and sound (senses) of the squirrel were integrated within the class object inside of "creatures.py" . The plotting specifically was done within the "plot_me()".

Within *snoo.py* the *creature_plot()* function would be called to plot the creature onto the correct set of axes along with sound and smell.

(2.2) The styling of the squirrel was done within *creatures.py* under the squirrel class object. Within the *plot_me()* function, the *patches* module of *Matplotlib* was used in order to create "*plottable*" shapes on the graph.

The logic behind this was to avoid creating complicated arrays that could theoretically serve the same purpose whilst having a high amount of detail.

(2.3) The normal movement algorithm of the squirrel is quite a robust one. The main component of the movement algorithm is located within the squirrel class object in *creatures.py*.

Within the movement algorithm lies a list of valid moves which contain coordinates movements. The relevant x and y coordinates are then added on to the *self.pos* variable.

The squirrel is allowed to move in the typical north, east, south, and west movements (Von Neumann). (COMP1005b, 2024)

Finally, this movement algorithm is translated into a viewable state in *snoo.py* where the function *creature_move()* is called.

(2.4) The movement colliders for the squirrel are quite simplistic whilst being sufficiently robust and easy to modify if needed. The collider system is located on lines 334 – 352 of the *step_change()* function in *creatures.py* under the squirrel class object.

By using the *self.pos* variable, the respective x and y coordinates of the squirrel can be sliced out and using if statements and some comparisons, the list of valid moves can be constructed accordingly to prevent the squirrel from moving into a location it should not be. i.e., inside the house, outside the fence, and outside the plot. Where any collider conditions are not met, the code allows for all possible moves...

(2.5) The health and age of the squirrel is implemented in *creatures.py* under the squirrel class object. On line 302, the age variable is assigned to the value of *self.age* and on line 308 the health of the squirrel is set to 100 under the variable *self.health*.

The function *health()* is used to return an integer value of *self.health*. Within *snoo.py* the function *creaturehealth()* calls the *healthloss* function for each creature in the list, which means that the squirrel's health is then calculated at each timestep.

(2.6) The squirrel also has the ability to interact with acorns that spawn on the map and consume them for additional health. Within *snoo.py* under the *_main_*() function, the number of acorns is defined to be 1.

The *acorn_plot()* function serves the purpose to assign the position of the to-be-spawned acorn to a random position in an accessible area by the squirrel.

In the *plot_treat_acorn_toy()* function, on line 641 the number of acorns is generated. A value of 4.6 set to the coordinate position of the treat in the array for the yard, this shows a light-brown colour on the main yard plot. Furthermore, the value of the smell is set to 10 for the coordinate position of the treat which is reflected on the smells plot as white.

Finally, on lines 656 and 657, the squirrel's movement system is altered to the alternative movement algorithm that allows it to move towards the treat.

Once the position of the acorn and puppy is the same, it is cleared off the map in the function *clear_acorns()*

(2.7) The movement towards the acorn is done through an adjustment to the movement algorithm that overrides the normal movement and allows the squirrel to directly go towards the toy.

Within *snoo.py*, once the toy has been plotted and *listname[1].move_towards_acorn = True*, within *creatures.py* under the squirrel class object in the function *step_change()*, the *move_towards_acorns()* movement algorithm is enabled as the condition for the if statement on line 322 is met.

(2.8) The consumption of the treat is done in a robust yet simple way. In *creatures.py* on line 306, when the position of the acorn and squirrel are equal, then the Boolean value for *self.acorn_collected* is set to *True*. This is done in the *move_towards_acorns()* function when the values of *dx* and *dy* are equal to 0, and then returned by the function *acorn_collected()*. This as a result runs *clear_acorns()*

(2.9) When the health of the squirrel falls below 50, the squirrel will stop moving and essentially go to sleep. In *creatures.py* on line 307, the Boolean value of *self.tired* is set as default to *False*.

In *snoo.py* under the *creaturehealth()* function, if the value of the health falls to a value < 50 on lines 769 and 770, the Boolean value of *self.tired* is set to *True*.

Back in *creatures.py* this is checked in the *step_change()* function on lines 124 and 125 where if the Boolean value is *True*, then the *sitdown()* function is active (line 326-327), this results in the position of the squirrel being fixed to its position when the value is < 50 for the health not allowing it to move.

(2.10) The squirrel also has the ability to stay away from the puppy. This is done through a modified collider system which considers horizontal and vertical distance from the puppy.

Within the *step_change()* function in the puppy class object, the variable *psx* is derived by taking the difference between the squirrels x position and the puppy's x position. The same is done to derive the variable *psy*, however unlike *psx*, the y difference is calculated using y coordinates.

* *Note that this system has some flaws that, as a result, mean that it does not function 100% of the time.*

Unfortunatley, due to time limitations, a robust system could not have been developed that would allow for the system to work 100% of the time.

(3.0) Human and Associated Features

The human is a major feature within the simulation, and it is also a requirement of the brief. The human is a visible object within the plot and serves to interact with elements on the plot using its senses which are: sight, sound, smell, and touch as well as with the squirrel and puppy.

(3.1) The plotting, position, smell, and sound (senses) of the human were integrated within the class object inside of "creatures.py" . The plotting specifically was done within the "plot_me()" function of the human, which when called into the function which appends it to the list allows for the plot axis and size limits to be defined.

Within *snoo.py* the *creature_plot()* function would be called to plot the human onto the correct set of axes along with sound and smell.

(3.2) The styling of the human refers to the design of the object, i.e., shape, colour etc. This was again done within *creatures.py* under the human class object. Within the *plot_me()* function, the *patches* module of *Matplotlib* was used in order to create "*plottable*" shapes on the graph.

The logic behind this was to avoid creating several complicated arrays that could theoretically serve the same purpose. It also allows for higher quality shapes and curves.

(3.3) The normal movement algorithm of the human is quite a robust one. The main component of the movement algorithm is located within the human class object in *creatures.py*.

Within the movement algorithm lies a list of valid moves which contain coordinates movements. The relevant x and y coordinates are then added on to the *self.pos* variable, where *self.pos* refers to the current position.

The human is allowed to move in the typical north, east, south, and west movements (Von Neumann). (COMP1005b, 2024)

Finally, this movement algorithm is translated into a viewable state in *snoo.py* where the function *human_move()* is called.

(3.4) The movement colliders for the human are quite simplistic whilst being sufficiently robust and easy to modify if needed. The collider system is located on lines 500 – 513 of the *step_change()* function in *creatures.py* under the human class object.

By using the *self.pos* variable, the respective x and y coordinates of the human can be sliced out and using if statements and some comparisons, the list of valid moves can be constructed accordingly to prevent the human from moving into a location it should not be. i.e., outside the fence, and outside the plot. Where any collider conditions are not met, the code allows for all possible moves...

(3.5) The human specifically has the ability to have the movement algorithm adjusted to allow it to go and place the acorn and treat once it reaches a certain position on the plot.

In *creatures.py* under the human class object, the function *step_ change_ treat_ drop()* checks the y coordinate of the human and compares it the position of the drop location which is the veranda of the house at position $y = 30$.

If the change in y, dy is > 0, then the valid moves that the human can make is only up (vertical move (-1,0)).

If the change in y, dy is equal to 0, then the position is equal to its current position.

In *snoo.py* under the *simulation()* function lines 881-890, the human is only allowed to move to drop the treat depending on the move frequency assigned by the user input *iteration.*

(4.0) Neighbor and Associated Features

The neighbor is a major feature within the simulation, and it is also a requirement of the brief. The neighbor is a visible object within the plot and serves to interact with elements on the plot using its senses which are: sight, sound, smell, and touch as well as with the puppy.

(4.1) The plotting, position, smell, and sound (senses) of the neighbor were integrated within the class object inside of "creatures.py" . The plotting specifically was done within the "plot_me()" function.

Within *snoo.py* the *creature_ plot()* function would be called to plot the neighbor onto the correct set of axes with smell and sound as well.

(4.2) The styling of the neighbor refers to the design of the object, i.e., shape, colour etc. This was again done within *creatures.py* under the neighbor class object. Within the *plot_ me()* function, the *patches* module of *Matplotlib* was used in order to create "*plottable*" shapes on the graph.

The logic behind this was to avoid creating several complicated arrays that could theoretically serve the same purpose. It was also done to allow for higher quality shapes and curves.

(4.3) The normal movement algorithm of the neighbor is quite a robust one. The main component of the movement algorithm is located within the human class object in *creatures.py.*

Within the movement algorithm lies a list of valid moves which contain coordinates movements. The relevant x and y coordinates are then added on to the *self.pos* variable.

The neighbor is allowed to move in only the horizontal direction on the footpath towards the right.

Finally, this movement algorithm is translated into a viewable state in *snoo.py* where the function *neighbor_ move()* is called.

(4.4) The movement colliders for the neighbor are quite simplistic whilst being sufficiently robust and easy to modify if needed. The collider system is located on lines 603 – 604 of the *step_change_right()* function in *creatures.py* under the neighbor class object.

By using the *self.pos* variable, the respective x and y coordinates of the human can be sliced out and using if statements and some comparisons, the list of valid moves can be constructed accordingly to prevent the human from moving into a location it should not be. i.e., outside the plot. Where any collider conditions are not met, the code allows for all possible moves...

(4.5) The neighbor specifically has the ability to throw the toy in a random position that is reachable by the puppy.

In *snoo.py* within the *plot_toy()* function, the number of toys is printed and using the *toy_plot()* function, a random position for the toy is generated to a list.

Then, the position is marked in the yard matrix to be 8 (Orange), the smell matrix to be 5 (50% intensity) and in the sound matrix 8 (80% intensity). This as a result allows the puppy to interact with the toy using sight, sound, smell and also touch.

In the *simulation()* function in lines 892 – 894, if the position of the neighbor in the x direction is equal to 30, and the counter is < 1, then the toy will be plotted. The counter is used to avoid unwanted subsequent plots of toys.

(5.0) The Plot and Associated Features

The main point of interaction with the simulation is with the plot. It is where all of the simulation elements are visible ranging from the humans, neighbors, creatures all the way to the senses of sound, smell and sight.

(5.1) The main plot is constructed using list slicing. From lines 75 – 106 of the *gen_yard()* function of *snoo.py*. The yard array is sliced and has values replaced from the original zeros array of the dimensions such that when it is plotted, the respective colours show up.

The colourmap used is *nipy-spectral* on the main plot.

In the function *plot_yard()* the axis is created with the respective annotations for the house, verandah, tree etc...

(5.2) The smells plot is used to create the senses of smell by showing the smell location on a colormap grey plot in *snoo.py*.

The *gen_smells()* plot returns the plan comprised of an array of zeros which correspond to the dimensions of the plot.

update_ smells() is a function that takes the current position of all creatures in the list and sets that position to be 10 which is 100% smell intensity, and it takes the previous position and sets the value to be 3 which is 30% intensity.

This function is called into the *creature_ plot()* function for which every timestep has the value updated.

Within the *human_ plot()* and *neighbor_ plot()* function the smells are also updated in the same manner as for the creatures.

In the function *plot_ treat_ acorn_ toy()*, the positional values of the acorn and treat have a smell attributed to them on the smells plot, and once collected the smells are then removed.

Finally, the *plot_ smells()* function plots the smells plot.

(5.3) The sounds plot is used to create the senses of sound by showing the sound location on a colormap grey plot in *snoo.py*.

The *gen_ sounds()* plot returns the plan comprised of an array of zeros which correspond to the dimensions of the plot.

update_ sounds() is a function that takes the current position of all creatures in the list and sets that position to be 10 which is 100% sound intensity, and it takes the previous position and sets the value to be 0 which is 0% intensity.

This function is called into the *creature_ plot()* function for which every timestep has the value updated.

Within the *human_ plot()* and *neighbor_ plot()* function the sounds are also updated in the same manner as for the creatures.

In the function *plot_ toy()*, the positional value of toy has a sound attributed to it on the sound plot, and once collected the sound is then removed.

Finally, the *plot_ smells()* function plots the smells plot.

(6.0) The Command Line and Other General Functions

The command line is an essential component of the program as is the main point of user interaction for the simulation. The command line allows the user to adjust parameters to their liking or to use the predefined default values which provide the user with a good understanding of the simulation.

Some general notes before function explanations:

To ensure the perpetual running of the simulation unless the user inputs the exit commands, the terminal incorporates the use of a while loop. This is a valid use case as otherwise, the use of "if" statements and other supplementary functions would result in less robust, less readable, and more complicated code with more points at which there could be technical issues. The use of the while loop results in a seamless and simplistic approach to these potential flaws.

Furthermore, for exception handling while loops and breaks have been used in a valid manner to allow for checks to only occur at the launch of the program allowing for more efficient code rather than having checks being performed at an unnecessary amount of time intervals.

(6.1) Exception handling for files and libraries is an important component of the code which ensures that all of the required code is present before the running of the simulation.

Exception handling for files and libraries has been integrated in lines 27 – 40 and 43 – 52. The exception handling that runs in lines 27 – 40 check for missing libraries such as *matplotlib*, *numpy* etc. On the other hand, exception handling on lines 43 – 52 check for missing modules from other files such as *creatures.py* and *colourtable.py*.

As mentioned in Table 1.1 – Traceability Matrix Section 6.0 and in the general notes above, the use of *while* loops, specifically *while True:* loops and *breaks* were essential to allow for exception handling to work.

This technically does not break the *one entry and exit point* (COMP1005, 2024) policy either as the main simulation itself does not utilise *breaks*.

(6.2) Exception handling for user inputs are an important component of the code which ensures that all of the user inputs are usable within all of the code and hence reduce the chance of any unexpected errors from occurring.

Exception handling for inputs has been integrated in lines 1033 – 1058. The use case of exception handing here is to ensure that all custom values inputted by the user are all integer values that are positive to ensure that the simulation does not crash when started.

As mentioned in Table 1.1 – Traceability Matrix Section 6.0 and in the general notes above, the use of *while* loops, specifically *while True:* loops and *breaks* were essential to allow for exception handling to work.

This technically does not break the *one entry and exit point* policy either as the main simulation itself does not utilise *breaks*.

(6.3) The user has two possible ways of running the simulation, one of them is with pre-set default values. This runs the simulation using the pre-defined

parameters which are stated in more detail in Table 1.0 – Default Parameters.

The simulation runs for 200 timesteps, with a plot time of 0.00s, a human and neighbor movement cycle value of 2 and a 30 timestep play time for the toy for the puppy.

(6.4) The other way the user can interact with the simulation is through adjusting the simulation to their liking using custom parameters and creatures/humans/neighbors.

The *Customisation Studio* is the main feature here in which the user is asked to define the following values. Number of simulation iterations (Timesteps), the time length of each timestep in seconds, the interval of timestep at which the human and neighbor move to place the treat, acorn, and toy and the amount of timesteps the puppy will play with the toy for.

In the character customisation section, the user is able to set the following parameter for the puppy, squirrel, human and neighbor: name, age and colour.

The user is presented with the option to show the table of acceptable CSS4 colours before selecting the colour as well. (Matplotlib n.d.) This code was sourced from *Matplotlib* documentation with appropriate references.

(7.0) UML Class Diagram

Note, + means that the method/variable is accessible by other functions whereas – means that only the class can access it.

Dashed lines shows that class has some dependency on others.

Showcase

For the showcase, to demonstrate three different scenarios, three different setup approaches have been taken.

For **scenario one**, the default values will be used. This includes, a 200 timestep duration, a 0 second timestep duration, the movement of the human and neighbor every 2nd timestep, and the play time of the toy with the puppy to be 30 timesteps. Furthermore, the default character settings will be used:

- A puppy with the name Snoopy, who is black and white, aged 74
- A squirrel named Sandy Cheeks who is orange and beige, aged 43
- A human named Charles who is black and bisque, aged 30
- A neighbor named Geoff who is red and bisque aged 35

This has been chosen to demonstrate the functionality of the program with no user changes to the default values.

For **scenario two**, a timestep setting of 600 will be used, a 0 second timestep duration, the movement of the human and neighbor every time step and the play time of the toy to be 50 timesteps. Furthermore, the character settings will be as follows:

- A puppy with the name Bluey, who is blue and white, aged 12
- A squirrel named Alvin who is red and white, aged 30
- A human named Tom who is white and bisque, aged 35
- A neighbor named John who is red and bisque aged 40

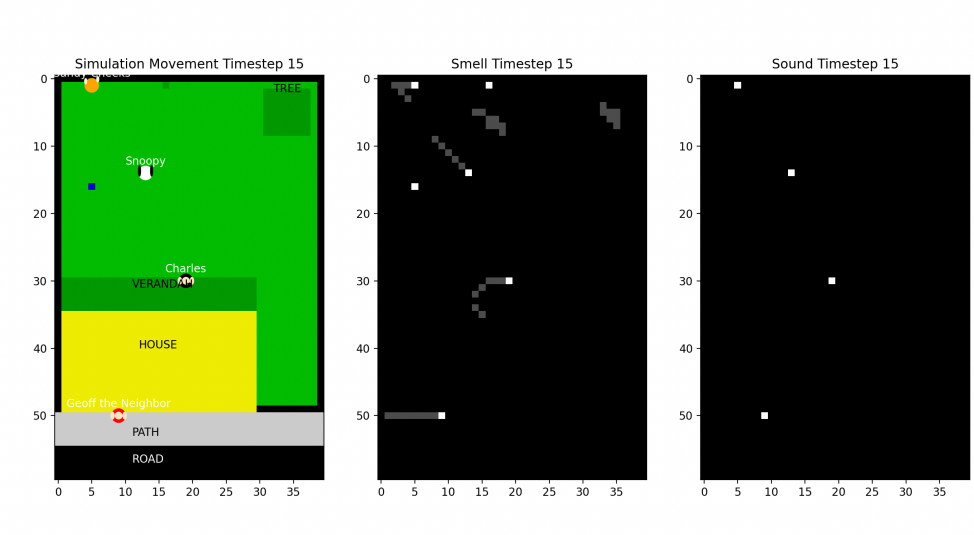
This set of options has been chosen to demonstrate the customisation ability of the simulation as well as have sufficient time to showcase the colliders, movement to the treat/acorn/toy. Furthermore, it will also demonstrate the ability for the puppy to play with the toy for a custom amount of timesteps.

For **scenario three**, a timestep setting of 40 will be used, a 1 second timestep duration, the movement of the human and neighbor every time step and the play time of the toy to be 10 timesteps. Furthermore, the character settings will be as follows:

- A puppy with the name John, who is orange and white, aged 12
- A squirrel named Tony who is Blue and white, aged 30
- A human named Johnny who is orange and bisque, aged 50
- A neighbor named Sam who is red and red aged 45

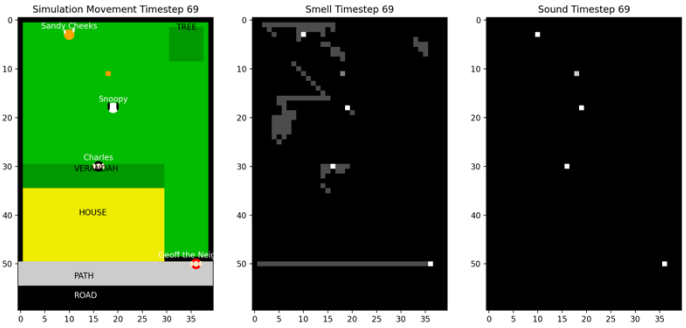
This set of options has been chosen to demonstrate the customisation ability of the simulation particularly the ability to accept a single colour and also to show that the simulation will always result in a random outcome.

Scenario One – Default Values



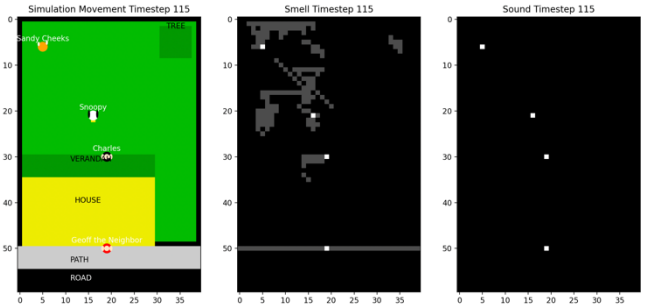
At timestep 15, it can be seen that the treat and acorn have been plotted and placed by the human, Furthermore, it shows that the puppy, squirrel, human and the neighbor are restricted to their boundaries via colliders for terrain.

This output also showcases the smell plot and sound plot highlighting how the smell decreases and the sound only being present at the position of the human. In the smells plot the treat and acorn also have their position visible.

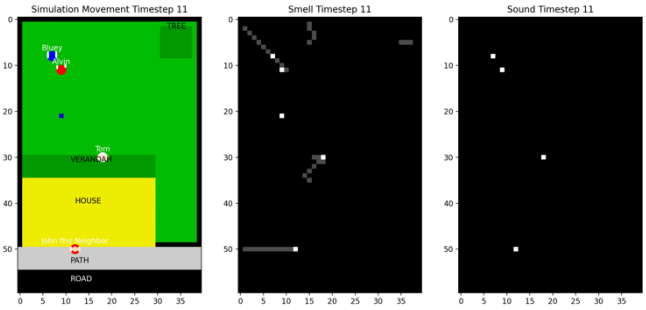


At timestep 69, the toy has been plotted as soon as the neighbor has its x position equal to 35, the toy is represented by an orange colour. The sound and smell of the treat also has been plotted on the graph.

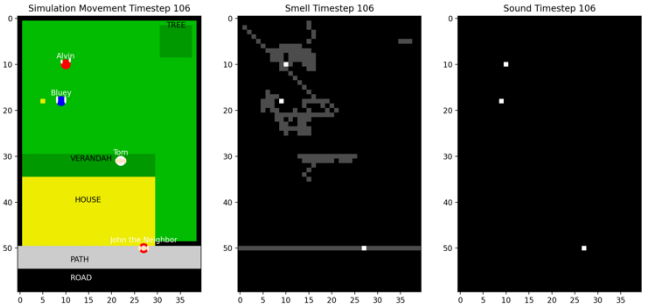
Just below the position of the puppy at timestep 115, it can be seen that the tile is set to yellow, which indicates that he has chosen to bury the toy. This showcases the random choice of bury or place and also shows variation in terrain. Overall, this scenario showcases that the simulation is able to function without any user intervention or any user parameter input.



Scenario Two – Custom Values (One)



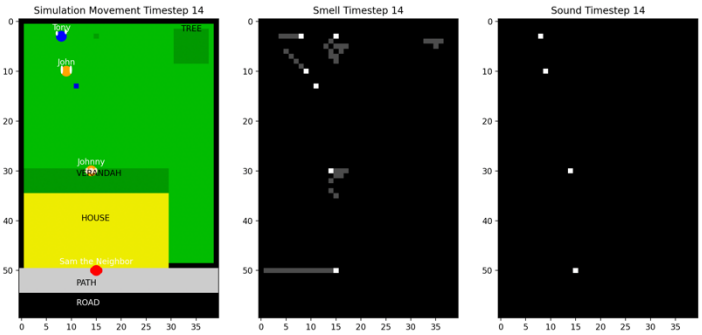
It can be seen in the above plots that the ability to customise characters done indeed work as intended as all characters have the respective colours and names applied to them on the plot. In this plot you can also see the treat on the map, and the smell map shows the movement of the puppy towards the treat.



It can be seen in the above plots that the ability for the puppy to hold the treat and then drop it at its prior location does work as after the timesteps that it held it, the treat was buried.

Scenario Three – Custom Values (Two)

It can be seen in the above plots that the single colour input did work with the neighbor, furthermore, this last simulation does show that the sequence of events does in fact vary over every single simulation resulting in a random output every single time.



Conclusion

Overall, this assignment has met the purpose of the brief which was combine all gathered knowledge developed over the course of this unit and cohesively combine it into an interesting and fun, application-based project to assess our understanding of the learnt concepts. All required elements of the brief were successfully integrated including animals humans, food sources/toys, senses, terrain and obstacles and collisions/interaction. The assignment code overall is flexible and robust with a fairly high level of user adjustment and usability through a command line. Finally, the assignment code follows the guidelines of PEP – 8. As the student developing the code, I had a considerable amount of fun undertaking this challenge. I am also fairly content with the final output as I believe that all of the effort, I put into this code fairly represents the assignment brief and, in some cases, goes beyond what is required.

Future Work

Further time could be spent into making a more robust movement algorithm with colliders that rely on grid searches around the position of creatures. This would significantly increase usability as then the plot could be scaled by the user rather than having fixed dimensions.
Another area of future work could be introducing more animals such as fish in a pond, a cat which could interact with the human and dog and maybe even another dog. These could all have the potential to reproduce and make offspring in a lifecycle. Further investigations could also be led into why the current squirrel/puppy avoidance algorithm only works sometimes rather than all the time.

References

COMP1005. "Fundamentals of Programming: Lecture 1" Accessed April 22nd, 2024, via BlackBoard

COMP1005a. "Fundamentals of Programming: Practest 3 Base Code" Accessed April 20th, 2024, via BlackBoard

COMP1005b. "Fundamentals of Programming: Lecture 5 Accessed April 21st, 2024, via BlackBoard

Matplotlib. n.d. "List of Named Colors — Matplotlib 3.4.2 Documentation." Matplotlib.org. Accessed April 28, 2024. https://matplotlib.org/stable/gallery/color/named_colors.html.

Curtin University. "Chicago 17th Author-Date." Accessed May 3rd, 2024. https://uniskills.library.curtin.edu.au/referencing/chicago17/introduction/