RoboViz

FINAL REPORT

To identify the behaviour of homogenous and heterogenous robot swarms, the RoboViz application was engineered. This software program allows an environment to be rendered as per user specification in order to contribute to swarm robotics research.

1. INTRODUCTION

The aim of this project was to design an application that could visualize modular robots in 3D space. To complete this task, the body and brain of a robot swarm was analysed and embodied using Python and Panda3D. Given the size of the robot swarm, this application visualizes either a homogenous or a heterogenous robot swarm. The body parts of the robots are rendered, connected, and visualized as per the given input files.

The environment in which the robots be rendered is specified in the input configuration text file, which must be read and parsed to execute a 3D scene of the task environment. The solution for this problem must produce a 3D visualization of the robot swarm that the user can pan, as well as zoom in and out of. These robots could appear in swarms or individually, consisting of several varying components and connections.

Stakeholders in this development were specifically researchers, who provided the files to be input to the application. They require this application to act as a tool for analysis of robot swarm behaviour. The client requesting this application will use it to contribute towards his research and projects being supervised. A Graphical User Interface was implemented to make the application more user-friendly

The SCRUM method of software development was followed, making this project agile and highly susceptible to changing user requirements.

1.1 Business Modelling

The first stage of this development process consisted of creating a business model, in which the value of the researchers' endeavours was analysed. The key processes of the stakeholders were modelled and documented to clarify the main purpose of the software.

1.2 Requirements

Possible use cases associated with the system were identified and analysed. Goals for the system, including performance and functionality, were outlined in this phase. In addition to this, the resources available to the SCRUM team were analysed to identify constraints and possible risks. This phase, ultimately, allowed the team to analyse the feasibility of the task at hand.

1.3 Analysis and Design

The object-oriented paradigm was used throughout the design of this application's architecture. This principle was useful in identifying, defining, and designing this application's classes and objects. The relationships between classes and objects were identified, as well as their implementation and interface, to provide a blueprint to the team for implementation. This strategy also eliminated a lot of miscommunications and allowed consensus to be reached sooner. In this phase, many UML diagrams were erected and a test plan outlined in order to guide the team's programming endeavours, as well as identification of edge cases and exceptions.

1.4 Implementation

The components of the software system were developed separately and integrated using GitHub. Due to the limited time provided in which to produce a finished product, evolutionary prototypes were created cyclically, with each prototype adding on more functionality to the prototype before it. Each sprint had a specific sprint backlog, which the SCRUM team tried to complete within the 7-day sprints to produce a working prototype.

The choice of evolutionary prototypes was driven by the team's reluctance to dispose of code after each sprint. With each shippable product, the application was displayed to the client and feedback requested to ensure the team's efforts were towards the specified target. Thus, vertical prototypes were made use of to ensure the client's most vital requirements were met as early in the development cycle as possible. This falls in line with the bottom-up approach of software design. The team prioritized constructing the codebase to reflect a focussed view of the most urgent requirements. Each time new features were implemented, they had only a reasonable amount of processing behind them. This allowed the user to navigate the system with full employment of those components whilst leaving space for easy accommodation of cosmetic changes that may be unearthed at a later point.

1.5 Validation and Deployment

A code freeze was implemented one week before the shippable product was set to be deployed. This allowed programmers to spend time testing the system to identify bugs, making the program more robust. Additionally, the level of code reuse was analysed, considering that systems are constantly changing, and in order for the software to remain valid, it should be easily maintainable. Lastly, a suitable license was chosen for the software application.

2. REQUIREMENTS

2.1 Functional Requirements

<table>
<tr><th></th><th>ACTOR</th><th>INPUT</th><th>OUTPUT</th></tr>
<tr><td rowspan="6">USE CASE 1

SPECIFY FILE PATHS FOR HOMOGENOUS OR HETEROGENOUS ROBOT SWARM</td><td rowspan="6">User</td><td>User calls command line interface by running the python script.</td><td>Case 1:
File not found or incorrect format
The program prints a relevant error message and exits.</td></tr>
<tr><td>The command line interface introduces a GUI, which will allow the user to browse their computer for 3 files.</td><td>Case 2:
Inconsistency between files
If there are inconsistencies affecting the way in which the robots are rendered, an error message is printed, and the program exits.</td></tr>
<tr><td>File 1:
The environment configuration text file (.txt)</td><td rowspan="3">Case 3:
All files correct
If there are no errors relating to any of the input file paths, the application will produce a window displaying the robots specified.</td></tr>
<tr><td>File 2:
The robot positions text file (.txt)</td></tr>
<tr><td>File 3:
The homogenous or heterogenous robot specifications file (.json)</td></tr>
</table>

	ACTOR	INPUT	OUTPUT
USE CASE 2 **VIEW REQUIRED FORMAT OF INPUT FILES**	User	The command line interface introduces a GUI, which will allow the user to press a "HELP" button.	A new window will display, instructing the user on how to make use of the application. File formats for each file type will also be specified.
USE CASE 3 **NAVIGATE 3D ENVIRONMENT**		**INPUT**	**OUTPUT**
		Use Case 1 is successful, and no errors are present in the inputted files. A new window is opened and the environment containing the specified robots is rendered.	The user will be able to perform many analysis techniques on the environment, as well as exit.
		The initial view will display all the robots in the environment. file. Each component of the robot will be labelled.	

2.2 Non-Functional Requirements

1. Performance

   To ensure the run-time of this application was kept brief, the codebase makes extensive use of object-oriented programming principles. Few, relatively large, classes and functions were instantiated, instead of small, fine-grained components. Critical operations were localized within these components, thus reducing the number of component communications. Files being parsed were read in only once. Arrays of objects were stored only once, and the traversal of these arrays was only performed when absolutely necessary. When multiple operations were to be performed with these arrays, such as storing and instantiating components, these two operations were completed in one traversal.

2. Security

   Security is not a critical requirement of this application as there is no personal information about the user stored. However, previous files generated by the user are stored in a .txt file, within a directory included with the application. This file is not protected or cleared on the user's computer; however, each user will have their own directory, which will not be shared.

3. Scalability

   Scalability is the measure of a system's ability to increase or decrease in performance and cost in response to changes in application and system processing demands. This application has been provisioned for scalability in terms of how well it withstands having to generate copious amounts of robots. Memory usage and comparisons within the code, as well as iterations, have been kept to a minimum in order for the program to be scalable to larger computations.

4. Availability

   Availability is the probability that a system is operational at a given time. Availability was a very important non-functional requirement for this application as it is imperative that when a user opens the command line interface, the application be always available for use. Therefore, the Event-Driven Architecture was implemented to place as much functionality in one core component as possible. This is much more reliable than the Layered Architecture, where there are multiple opportunities for the program to fail. However, with this implementation, there are only three classes that control the operation of the application. If one of them fail, which has not proven to be likely, then the whole system fails.

5. Maintainability

   To ensure the maintainability of the software, the system architecture was designed to be fine-grained with self-contained components that are readily susceptible to change. Thus, the software modules were kept highly cohesive with low coupling between the classes. Team

members were highly aware of the distribution of their code, and thus consistently commented their code and produced documentation.

2.3 Usability Requirements

These requirements involve the software being acceptable to users, allowing the code to be more than just usable, but additionally user-friendly. To make this application as attractive as possible, a GUI was composed. This GUI resembles many applications for the input of files, which avoids the extra effort on the user's side of trying to understand how to use the file input system. Thus, the interface makes use of intuition, with the main goal being to allow easy navigation. This broadcasts a low perceived workload to the user, avoiding frustration that sometimes comes along with a new application.

2.4 Use Cases

2.4.1. Enter file paths

The actor can open the application by typing in the path of the python program to the command line interface. If the user does not want to use the GUI, they are able to enter the file paths to the CLI in the following order: <configuration text file> <positions text file> <robot json file>

Alternatively, the user is able to select specific files from their computer using the 'BROWSE' button on the GUI.

Alternative Path: The User Presses 'EXIT' and the program closes.

2.4.2. Request Help

There are two 'HELP' buttons included in this application: one on the main screen of the GUI, as well as an identical one on the 'BUILD' screen. Upon pressing one of these buttons, a new window will open, which provides each of the following explanatory functions:

1. Describes to the user, in-depth, how the application can be used.
2. Provides an example of each of the formats of the input files.
3. Provides a link to the application's documentation for further inspection of certain properties.

Alternative Path: The User Presses 'EXIT' and the program closes

2.4.3. View Robot Swarm

Upon pressing the 'SUBMIT' button, the user may view the robot(s) their input files have generated. The scene assembled depends on the following conditions:

1. The JSON file input contains the *swarm* label and specifies there to be more than one type of robot. This will render a heterogenous robot swarm.
2. The JSON file does not specify there to be more than one type of robot, however the positions file input by the user specifies more than 1 robot. This will render a homogenous robot swarm.
3. The JSON file specifies only one type of robot and the positions file only indicates one robot in the swarm. This will render only one robot.
4. The user chooses the "auto-pack" option. This will disregard the positions file chosen by the user and auto-position the robots in their JSON file. The environment may also be re-sized if needed.

Alternative Path: There are many possible sources of inaccuracy at this stage of the process. The user can make one of the following mistakes:

<table>
<tr><th>INACCURACY</th><th>GUI RESPONSE</th><th>CLI RESPONSE</th></tr>
<tr><td>One of the required files not provided</td><td rowspan="2">When the user tries to press ‘SUBMIT’, red error text will display, informing the user of their mistake.</td><td rowspan="2">When the user submits their file paths via the command line interface, a message stating there mistake will be returned.</td></tr>
<tr><td>One of the files is of the incorrect format</td></tr>
<tr><td colspan="3">INCONSISTANCIES BETWEEN FILES</td></tr>
<tr><td>The number of robots in the heterogenous swarm (.json) does not correspond to the number of robots in the swarm specified by the positions file/configuration file.</td><td rowspan="2">When the user tries to press ‘SUBMIT’, red error text will display, informing the user of their mistake.</td><td rowspan="2">When the user submits their file paths via the command line interface, an [ERROR] message will be displayed, describing their mistake.</td></tr>
<tr><td>The number of robots in the swarm as specified by the configuration file does not correspond to the number of robots in the swarm specified by the positions file.</td></tr>
</table>

2.4.4. Build Robot

Upon pressing the ‘BUILD’ button, the user may add components to a simulation, specifying the type of component, as well as its ID. Additionally, the user may specify the source slot and the orientation of that source slot. Additional functionalities include:

1. Saving the specified file to a directory as specified to the user
2. Importing a pre-constructed JSON file into the directory and making changes to the components listed

Alternative Paths:

1. The user tries to add the same component twice
2. The user tries to add a component with an ID containing a space
3. The user trues to build a robot with no components/only one component
4. The user incorrectly specifies the source/destination slot between components
5. The user inputs a space in the specified directory

For all the above cases, a relative error box will be displayed on the GUI informing the user of their mistake.

2.4.5. Navigate 3D Environment

Once the actor has filled in all required information and there are no debilitating errors, pressing the ‘SUBMIT’ button will open a new window containing a 3D environment. The scene allows the user to perform the following functions:

1. Select a robot or a specific component, as well as view the ID of each.
2. Zoom in and out of the rendered environment.
3. Pan and rotate the environment.
4. Switch the camera focus to view different pieces of the rendered robot(s).
5. Toggle component labels.
6. Exit the environment, which will take the user back to the initial RoboViz screen.

Alternative Path: An ‘ERRORS’ window will display before the 3D environment is rendered, informing the user of either of the following occurrences:

1. If there is a collision between two or more robots, such as they are being rendered atop each other or their components make contact, the error window will inform the user of which robots are possibly colliding.
2. If a robot has its components exceeding the bounds of the environment, the error window will inform the user which coordinate is faulty.

These errors, however, will not prevent the user from viewing their chosen environment.

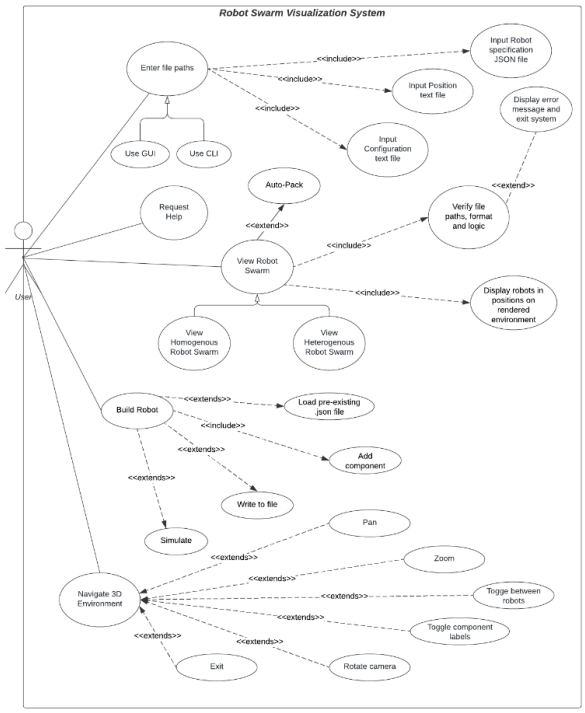


Figure 1 *Use Case Diagram*

2.5 Overview of Analysis Artefacts

In addition to the Use Case Diagram, an Analysis Class Diagram (Figure 4) and Architecture Diagram (Figure 5) were produced. These diagrams will be presented in the Design Overview section.

To understand the context of this application, a Sequence Diagram (Figure 2) was erected. This diagram outlines the basic, initial requirements of the system. It represents the user entering the requested files to create and render a robot. As meetings with Product Owners progressed, more functionality was added to the system.

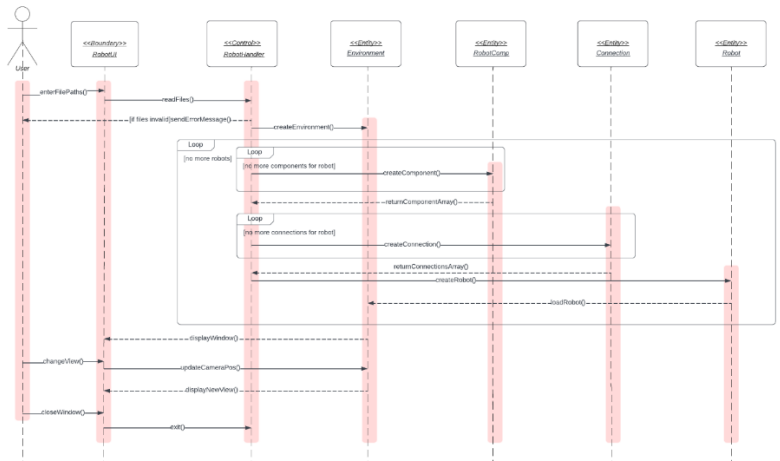


Figure 2 *Sequence Diagram*

Furthermore, to gain a conceptual view of the system as design progressed, an analysis class hierarchy diagram was produced. As more classes and methods were added to the codebase, the document was updated to produce the final artefact displayed in Figure 3. This diagram expresses the relationship between classes, as well as the functionality each class inherits from other classes.

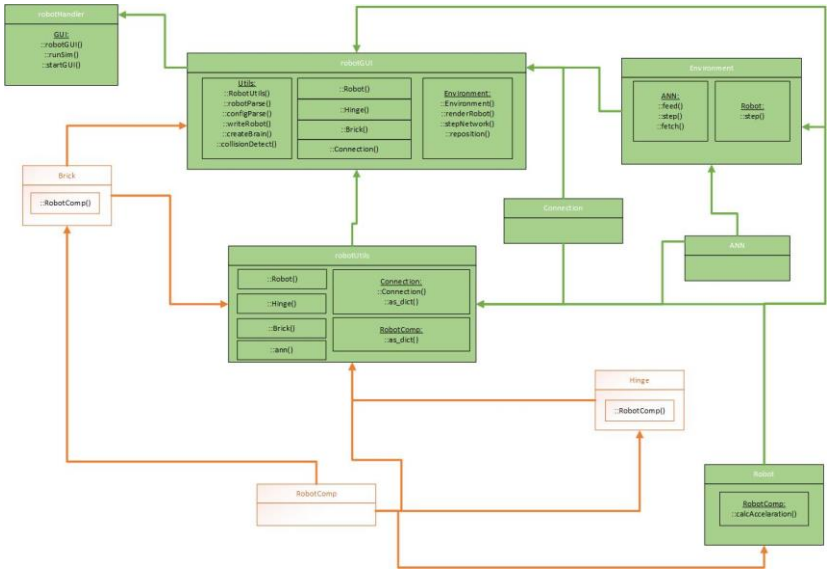


Figure 3 *Analysis Class Hierarchy Diagram*

3. DESIGN OVERVIEW

3.1 System Behaviour

Figure 4 *Design Class Diagram*

The Design Class Diagram in Figure 4 describes the basics behind the behaviour of the RoboViz application. RobotHandler initializes the RobotGUI based on whether or not the user is running the application via the GUI or CLI. RobotGUI and RobotUtils have a one-to-one cardinality, as RobotUtils is a logic-based class that controls file and user I/O. Within this class, many Robot objects may be created. A robot object may only have one ANN.

Hinges and Bricks act as the building blocks for this application. They are subclasses of the RobotComp abstract class. This parent class is made use of extensively in Connection objects, as one connection consists of two components. Connections are combined to construct a Robot object. There is only one Environment object present for a single run of this program, which may render zero or more Robot objects.

3.2 System Architecture

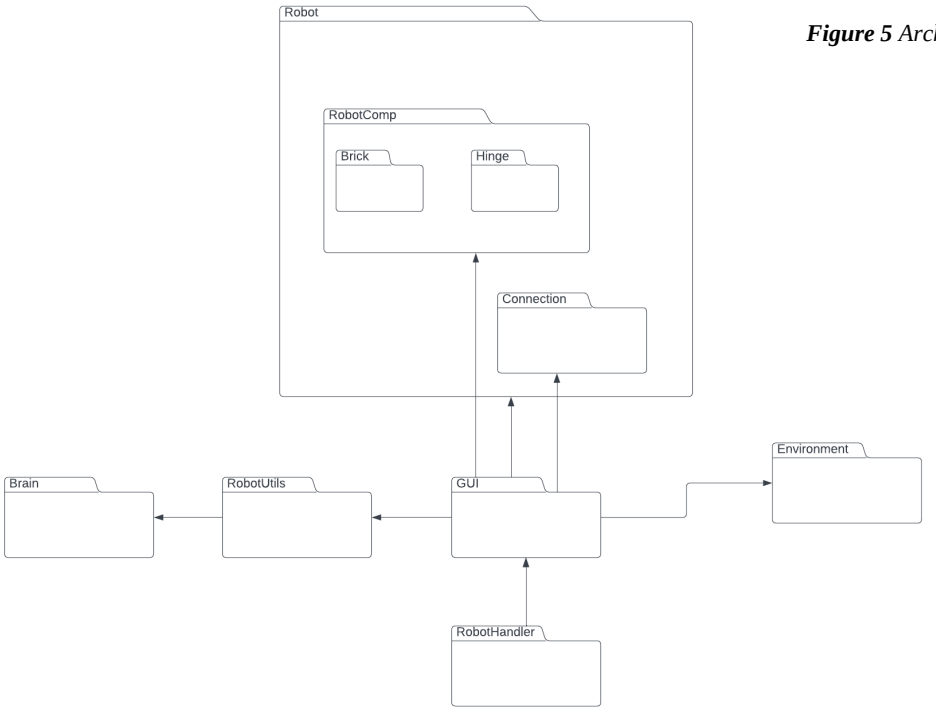


Figure 5 *Architecture Diagram*

The architecture used for our program is an Event-Driven Architecture. Where the Command Line Input and Graphical User Interface act as event producers.

The following class act as event routers:

a) RobotUtils
b) Brain
c) Robot
d) RobotComp
e) Connections

The Environment class acts as the event consumer.

3.3 Principal Algorithms

3.3.1 Robot Rendering

A robot rendering process was decided upon that is based on the connections between the different components. Instead of looking at each component in the Robot, a List of the Robot's connections would be stored in the Robot object and iterated through to render each component.

Only the destination component is rendered with each iteration because the source component has already been rendered as the destination component in the previous iteration. The source component is used to calculate the position of the destination component.

The basic flow for this is described in Code Listing 1.

This was considered to be better than looping through all the components in a robot, as one component could be the parent for multiple other components. Therefore, a List or similar data structure would have to be maintained for each component's children. With this method, only one List of Connections would need to be kept for each Robot.

```
for connection in Robot connections
  if connection.src is root component
    src = load src model (CoreComponent)
    src.setPos = Robot core position
    reparent src to robotNode          # robotNode = parent node to all Robots

  dst = load dst component
  dst.setPos = pos relative to src     # calcPos(src, dst, connection) returns pos & heading
  dst.setHeading = heading relative to src

  # apply roll (orientation) to hinges
  if dst is a Hinge:
    dst.orientation += src.orientation    # add parent's orientation (relative)
    # rescale to {0, 3}
    while dst.orientation > 3
      dst.orientation -= 4
    dst.setRoll = dst.orientation

  reparent dst to src
```

Code Listing 1 *Creation of a Connection Object*

```
# if connected to a brick, reduce dims to slot hinge into groove
if src or dst is a Brick:
  src_dims -= buffer_value
src_slot = src_slot - src.direction                # get actual 'global' src_slot based on src
comps direction (heading)
if src_slot < 0:                                   # scale src_slot back to {0, 3} if negative
  src_slot += 4
heading = src_slot + dst_slot                      # get heading of dst component
dst.direction = heading
dst.setHeading = heading

# determine which axis of dimensions to use (0 or 2 = y, 1 or 3 = x)
if src_slot == 0 or 2
  src_dim = src_dims[1]
else
  src_dim = src_dims[0]
if dst_slot == 0 or 2
  dst_dim = dst_dims[1]
else:
  dst_dim = dst_dims[0]

# determine in which direction the dest. component should be shifted to appear connected

                    ector(0, -(src_dim + dst_dim), 0)
elif src_slot == 1
  dst_pos = src_pos + vector(-(src_dim + dst_dim), 0, 0)
elif src_slot == 2
  dst_pos = src_pos + vector(0, src_dim + dst_dim, 0)
elif src_slot == 3
  dst_pos = src_pos + vector(src_dim + dst_dim, 0, 0)
```

Code Listing 2 *Calculation of Component Position*

3.3.2 Component Position Calculation

The positions and headings of components being placed in the scene needed to be calculated based on the position and dimensions of the parent component and the slots associated with the Connection. Furthermore, the source slot numbers of a component needed to be changed depending on the heading that it was placed at.

The basic flow of this is described in Code Listing 2.

This algorithm was used as it takes the changing of slot numbers based on the heading of a component into consideration. It also does not use any hardcoded values and so is easily evolved to deal with any new components that may need to be added.

3.4 Data organization

3.4.1 Class Data Organization

The required data was broken down into 3 object classes: RobotComp, Connection, and Robot. These represented the data stored in the robot JSON file. By having these objects formatted according to the JSON file, an intuitive understanding is enabled between knowing how a robot is assembled in the JSON file and knowing how it's assembled in the software and 3D environment.

RobotComp: This class closely relates to each entry in the "part" section of the JSON file. Each field is stored in a class attribute of RobotComp.

```
"part": [
        {
          "id": "Core",
          "type": "CoreComponent",
          "root": true,
          "orientation": 0
        },
]
```

Code Listing 3 *Dictionary representation of Component Object*

```
"connection": [
        {
          "src": "Core",
          "dest": "Hip1",
          "srcSlot": 0,
          "destSlot": 0
        },
]
```

Code Listing 4 *Dictionary representation of Connection Object*

Connection: Each entry in the "connection" section of the JSON file is extremely alike. Each field is stored in a class attribute of Connection and the 'src' and 'dest' attributes are stored as RobotComp objects.

Robot: This class consists of a List of RobotComp objects and a List of Connection objects which closely relates to how one robot in the JSON file is made up of many parts and connections.

3.4.2 Data Storage

Input files consist of a configuration text file (environment & swarm size), robot positions text file (x, y, z coordinates of the robot cores) and a robot JSON file (specifications of one or many robots).

Theoretically, the user may browse for these files in any location on their drive, but the recommended storage spaces are 3 directories in the root project directory: *./config/*, *./positions/* and *./json/*. This is also where the sample files are located. This enables the easy management of different swarm configurations.

Output files consist of a LastRender text file (the last used file paths) and robot JSON files generated by the robot builder The LastRender.txt file is generated by the RobotGUI class whenever a user loads a simulation through the GUI. It contains a list of the 3 file paths to the configuration, positions, and robot files that they used. When the GUI is next loaded, these will be restored to the text boxes so that the user does not have to constantly enter the file paths.

Custom robot JSON files can be generated by the robot builder GUI mode. If the user has the option selected, it will write their built robot to a JSON file to the *./json/* directory with the correct structure and custom name.

4. IMPLEMENTATION

4.1 Data Structures

Table 1 *Summary of Key Data Structures*

Name and Data Type	Data Structure
connections:Connection	List
robot_pos:LVector3f	Dictionary
SceneGraph:PandaNode	Tree
robotArr:Robot	List
labels:textNode	List

4.1.1 Connections

The *connection* data structure is implemented as a 1D Python List, a dynamic array, and is used to store the Connection objects which make up one Robot. This is iterated through in the Environment method, *renderRobot(…)*, in order to render a Robot down from the Core Component.

A List/dynamic array was chosen for the *connections* data structure as the only operation performed on it would be appending an item to the end of it. This operation would occur whenever a new Connection was read in from the Robot JSON file). Appending has a time complexity of O(1). Similarly, a Connection in the middle of the data structure may have to be accessed and a List has a time complexity of O(1) for finding an item. The afore-mentioned events are described in Figure 6.

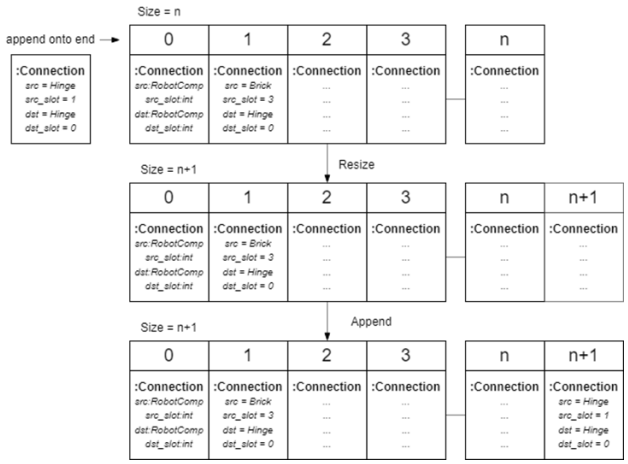


Figure 6 *Connection Data Structure*

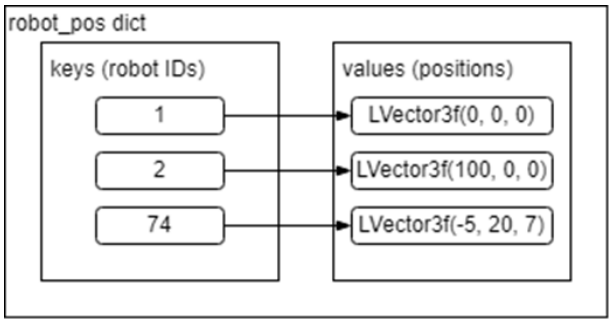


Figure 7 *Robot Positions Data Structure*

4.1.2 Robot Positions

The *robot_pos* data structure is implemented as a Python Dictionary, through the use of a hash table. It is used to store the position of each Robot in the scene, enabling the user to switch camera views between them. These positions are stored as three-component LVector3f vectors which are the standard in Panda3D for recording position in a 3D environment.

A Python Dictionary was chosen for this data structure as it would be accessed to cycle the camera's view through each Robot position (*switchFocus* method), as well as to access the position of a specific Robot based on its ID.

A List, or similar array-based data structure, would have been adequate for fulfilling the first task. However, the dictionary is effective for the second task as it allows the simple association of a Robot's position with its ID. It is also efficient, having an average case time complexity of O(1) and a worst-case time complexity of O(n) for getting a Robot's position. The *robot_pos* dictionary data structure can be represented by Figure 7.

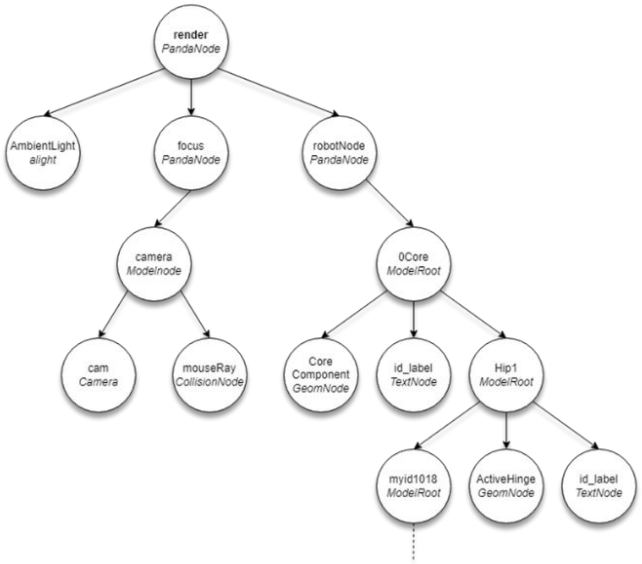


Figure 8 *SceneGraph Tree*

4.1.3 Scene Graph

The Panda3D *SceneGraph* is implemented as a Tree data structure and is used to store all PandaNodes in the Scene. This includes ModelNodes, GeomNodes and LightNodes and the parent node, *self.render*, among others. Importantly, it also stores the hierarchical relationship between these nodes. In a single Robot, the Core component would be the parent, and all other models would then be parented to the source component in their relevant Connection.

The *SceneGraph* is implemented as a Tree by Panda and the developer only shapes the Tree's hierarchical structure. The Tree structure is especially useful for building modular robots as different components are parented to others in the connections laid out in the JSON files. This is also partly why Panda3D was chosen as the rendering engine for this project. A SceneGraph Tree is represented in Figure 8. This visual representation, however, does not display certain in-build Panda3D node branches.

4.1.4 Robot Arrays and Labels

The *robotArr* data structure is implemented as a Python List and is used to store each Robot (constructed from the JSON file) before it is passed into the Environment *renderRobot* function. It is used for a short time and only ever appended to.

The *labels* data structure is also implemented as a Python List and is used to store the labels (TextNodes) of each component in the Scene. It is only ever appended to (when creating new labels) or iterated through (when toggling labels on or off).

A Python List, because of the reasons described for the *connections* data structure, was chosen as adequate for both data structures.

4.2 User Interface

4.3.1 Basic Functionality

The User Interface was implemented to be simple but effective, as the program will be mostly used for research purposes. The main goal was to create a system that required as little work from the user as possible, thus the user must only rely on intuition. PySimpleGUI library was used in the creation of the User Interface for RoboViz. Additionally, as the SCRUM team wanted to place more focus on the implementation of certain functionalities, PySimpleGUI was chosen due to its ease of use.

The GUI boasts 3 input text boxes accompanied by 3 BROWSE buttons, features which allow selection of files needed to start the robot rendering. Compared to using the command line, this is much easier and promotes accuracy, when compared to using CLI. The probability of the user inputting incorrect file paths is drastically diminished. However, to further reduce the possibility of confusion, a HELP button was included, providing an explanation to the user on how to use the user interface.

After selecting 3 files, the user may click the SUBMIT button and the robots in the provided JSON will be rendered. If there are any collisions, the user will be notified through the user interface. Marking the AUTO-PACK check box automatically renders the robots into the environment, ensuring there are no logic errors. Furthermore, the BUILD button allows the user to build their own robot in another window and then produces a JSON file after they created a robot. Clicking EXIT will terminate the program.

4.3.2 Panda3D

The user interface for the viewing window User Interface, in which the robots and environment are rendered, consists of the following elements:

4.3.2.1 HELP DISPLAY

The help display incldes a list of helpful key binds. They are visible when the viewing window opens and throughout the program but can be hidden with ‘H’.

4.3.2.2 SELECTION DISPLAY

The robot and component selection display visualizes the IDs of the currently selected robot and component. It is visible when the viewing window first opens and throughout the program.

4.3.2.3 LABELS

The Robot and component labels display the IDs of the robots and components in the scene. They are invisible when the viewing window opens. Visibility toggled by the user clicking ‘L’. When clicked on, the labels are enlarged.

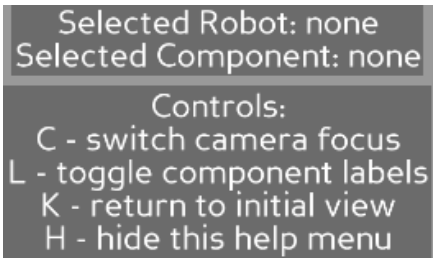


Figure 9 *Help Menu*

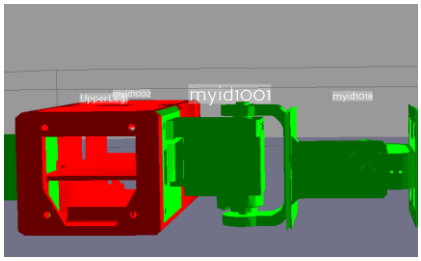


Figure 10 *Selection Display*

The help and selection displays were implemented using Panda3D’s UI library, DirectGUI, via the implementation of an OnscreenText object. This provides a simple way to render 2D text onto the Panda3D scene.

```
sel_text = 'Selected Robot: none\nSelected Component: none'
self.sel_textNode = OnscreenText(text=sel_text, pos=(1, 0.8), scale=0.04,
                                 fg=(1, 1, 1, 1), bg=(0.3, 0.3, 0.3, 0.6),
                                 align=TextNode.ACenter, mayChange=1)

self.accept('h', lambda: self.help_textNode.show() if self.help_textNode.isHidden() else
self.help_textNode.hide())
```

Code Listing 5 *Selection Display Implementation and Hep Display Toggling*

The robot and component labels were implemented using Panda3D’s TextNode objects, which allow them to be repositioned and rescaled at whim. They were set as ‘billboards’ so that, although they are 2D, they will always rotate to face the camera.

```
label = TextNode('id_label'
label.setText(text)
label.setAlign(TextNode.ACenter)
label.setCardAsMargin(0, 0, 0, 0)
label.setCardDecal(True)

self.text3d = NodePath(label)       # add text to node
self.text3d.setScale(3, 3, 3)
self.text3d.setTwoSided(True)
self.text3d.setBillboardPointEye()  # make text billboard (move with camera)
self.text3d.reparentTo(parent)
self.text3d.setPos(self.render, pos + LVector3f(0, 0, 20))
```

Code Listing 6 *Component Label Implementation*

4.4 External Libraries

4.4.1 Panda3D

Panda3D is an engine for 3D rendering and game development. It is the basis for the entire RoboViz project and is used to render each robot and the environment in which they are placed. As it is built in

C++, it is quite lightweight and performant. Additionally, it interfaces with Python, making it simple to learn and use. Panda3D also provides deployment tools, allowing the easy compilation of Python applications into platform-specific binaries (Windows, Linux, macOS).

Panda3D is licensed under the Modified BSD License, meaning that the source code is open, and no costs must be paid in distribution of a product developed with Panda3D. Note that the version of Panda3D listed for use in RoboViz is version 1.10.11. This is currently not the latest build, but it ensures that the program will work as expected. See the User Manual for more information.

4.4.2 PySimpleGUI

PySimpleGUI is a simple UI framework which simplifies the usage of popular Python UI libraries such as tkinter, Qt, and WxPython. All UI elements outside of the Panda3D window are built using it. RoboViz does not call for extravagant UI elements, and this library allowed much simplicity. It allowed a swift set up of a functional UI with not much code.

PySimpleGUI is licensed under the GNU Lesser General Public License, meaning that PySimpleGUI can be incorporated into any software, including proprietary.

4.4.3 rectangle-packer

rectangle-packer is small library for solving the packing problem, i.e. given a set of rectangles with fixed orientations, find a bounding box of minimum area that contains them all with no overlap. It is used in the auto-pack feature, to automatically determine non-overlapping positions for any number of homogenous or heterogenous robots in a scene. It is licensed under the MIT License, meaning that it can be incorporated into any software, including proprietary.

4.4.4 Other libraries

NumPy, a library containing a large amount of scientific computing utils, as well as SymPy and SciPy, for symbolic mathematics, were particularly useful in the construction of the ‘brain’ of the robot. All 3 use the Modified BSD License.

4.5 Analysis of Classes

4.4.1 RobotGUI

The RobotGUI class encompasses the functionality displayed in Figure 11.

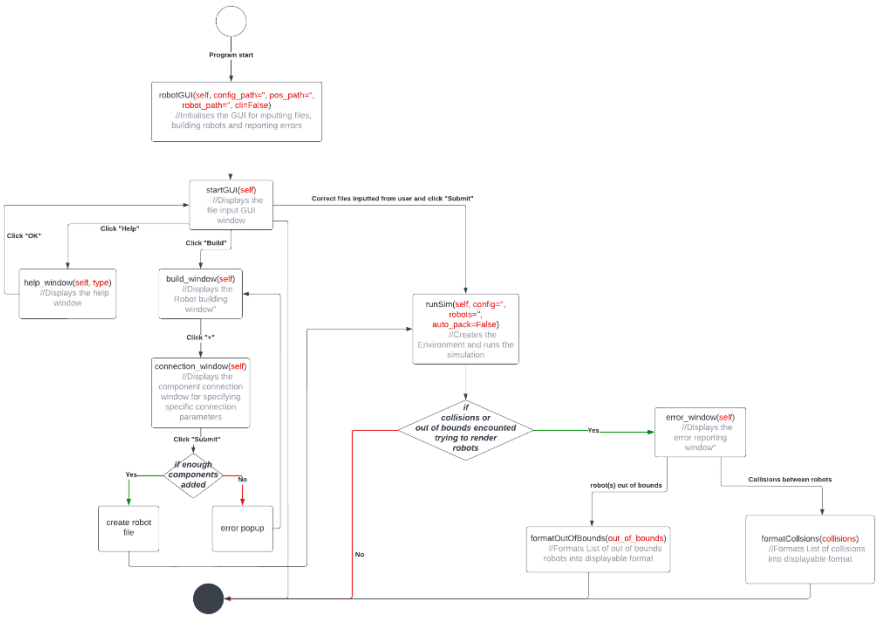


Figure 11 *RobotGUI Algorithms*

Perhaps the most vital method to this class is the *startGUI()* method. It creates and destroys the starting menu that the user will interact with to parse in files, under the premise that they are not using CLI. Figure 12 displays how the method first checks if the LastRender.txt file exists, and creates the starting menu with or without the LastRender.txt lines filled in.

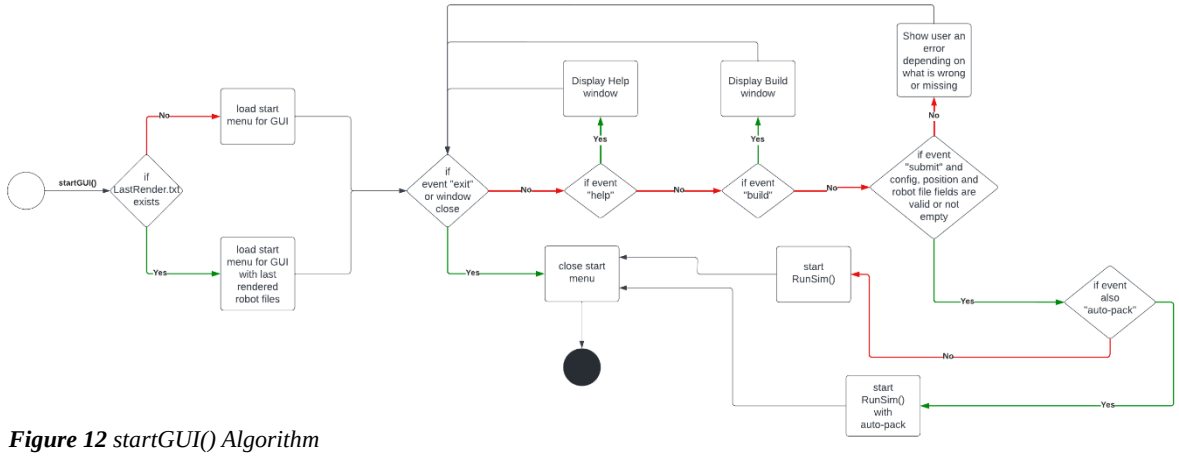


Figure 12 *startGUI() Algorithm*

4.4.2 Environment

i. Rendering a Robot

This algorithm is vital to the visual display of this application. It accepts a Robot object as the only argument and loops through its connections List to render each component in the Panda Scene. The output of this is one complete, visible Robot.

It is called in a loop from the RobotGUI *runSim*() method so that each Robot in the swarm is rendered. The algorithm for this method is represented in Figure 13.

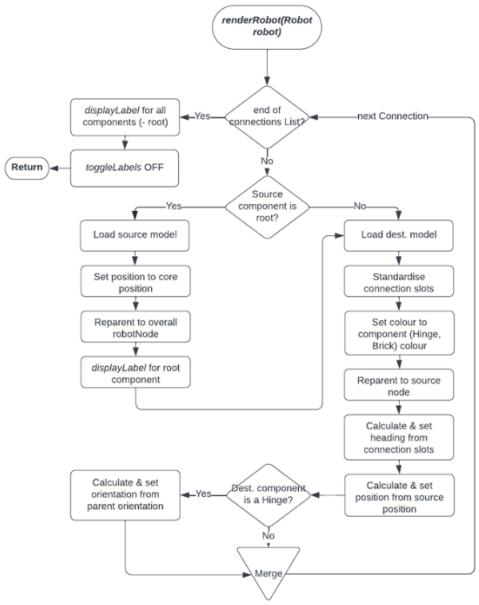


Figure 13 *renderRobot() Algorithm*

This method determines which component and Robot the user has selected when they click on a specific component in the Panda Scene. Panda generates a List of NodePaths that are closest to the mouse cursor and a simple sort reveals the closest one.

Any selectable Nodes are pre-tagged with the ‘robot’ tag to indicate that they can be selected. Only these are considered when finding the nearest component to the mouse click. The algorithm for this method is displayed in Figure 14.

This method is initiated by the Panda ‘accept’ statement which listens for the ‘mouse1’ event.

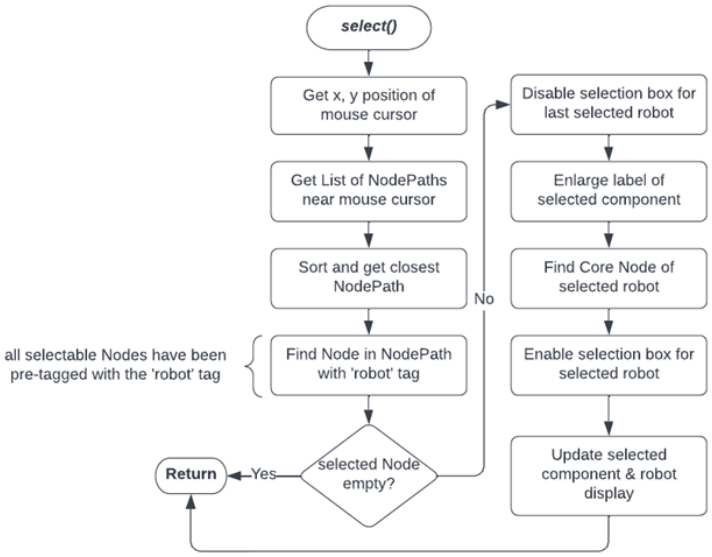


Figure 14 *select() Algorithm*

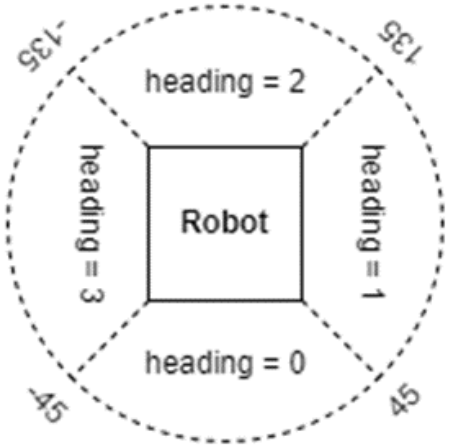


Figure 15 *Robot Orientation Quadrants*

iii. Repositioning a Robot

The *moveRobot*() method determines which way the camera is facing relative to the selected Robot. It moves the selected Robot according to which arrow key is pressed. This allows for the intuitive movement of Robots wherever they are in the scene. It takes in only one argument – the direction – which specifies how the Robot should be rotated relative to the “camera”.

The *moveRobot*() method allows the Robot to be moved left when the left-arrow key is pressed. Likewise, pressing the up-arrow key will move the Robot forward and so on. Without this system, movement direction would remain constant and become unintuitive no matter the direction the camera.

For instance, if the camera were looking at the back of the Robot, pressing the left-arrow key would move the Robot right. If the camera was facing the right side of the Robot, pressing the left-arrow key would move the Robot forward.

Instead, this method separates the space around a Robot into 4 quadrants as shown in Figure 15. The direction of the camera is then evaluated to determine which quadrant it falls into, and the direction of movement is changed by:

```
direction = int(direction) - heading
```

Where heading is the orientation of the component. The shift is then applied.

This method is called from a Panda ‘accept’ statement which listens for any arrow key event.

4.4.3 RobotUtils

This class acts as the interface between the RobotGUI and Environment classes. The RobotGUI class accepts information from the user and passes it to the RobotUtils class. This class will either output

usable numerical data or data structures to be passed back to the GUI for input to Environment, or it will produce an error for incorrect provision of information. An overview of the structure of this class can be found in Figure 16.

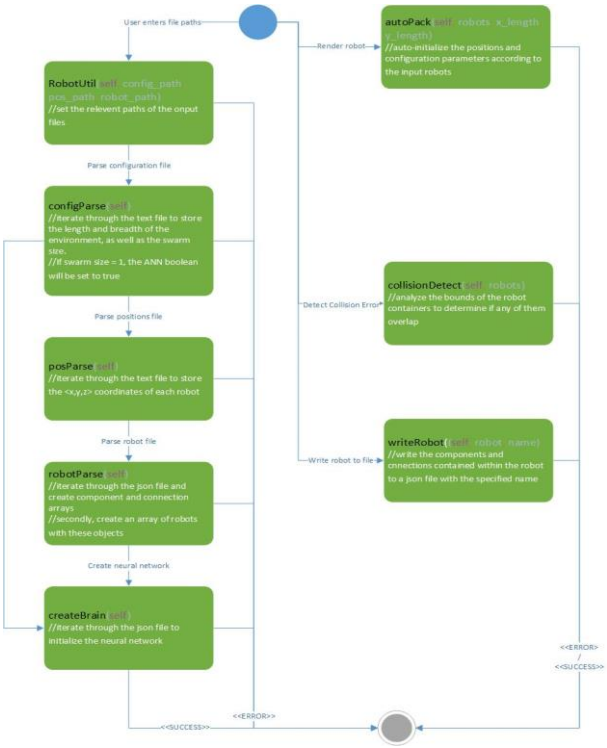


Figure 16 *Overview of RobotUtils Algorithms*

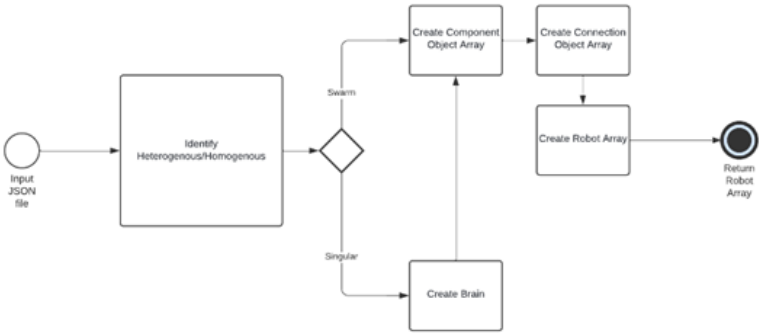


Figure 17 *robotParse() Algorithm*

i. Creating Robot Objects

The *robotParse*() algorithm is represented in Figure 17. It represents the methodology used for parsing a JSON file and storing the information obtained in relative objects. The *createBrain*() algorithm, as well as the *configParse*() and *posParse*() methods follow a very similar structure.

In order to write Robots back into a JSON file, this algorithm was reversed. To accurately perform these tasks, the *json* encoder/decoder API was made use of so JSON files could be parsed correctly. Additionally, the *copy* python library was utilized so that deep copies could be made between source and target objects.

ii. Auto-Packing Robots

If the user indicated on the GUI that they required the Robot objects to be auto-packed into the environment, this method analyses the bounds of each Robot in the array. Based on these metrics, each Robot is assigned a ‘rectangle’ with a PACK_BUFFER, and these (width, height) coordinates are added to an array. Then, using the *rpack* library, these Robots are “packed” into the environment with no overlaps. This function also endeavours to keep the enclosing area as small as possible. This “packing” is then returned to the GUI, however if there is a “PackingImpossibleError”, the relative error will display.

iii. Collision Detection

A nested for loop was used to traverse through the Robot array and analyse the x, y and z bounds of each robot. If the bounds of these robots were found to cross, the *id* of each colliding robot was added to a *collisions* array and returned to the GUI for indication to the user.

4.4.4 Robot

This class holds all robot components and connections associated with a specific robot, as instantiated by RobotUtils. It is responsible for the feeding the physical representation of the data to the Environment class for output to the user. An overview of the algorithms contained within this class is displayed in Figure 18.

i. Creating Bounds

Detecting whether or not Robot objects are rendered atop each other, or beyond the border of the environment, is a 3-step process. This process is displayed in Figure 19. Firstly, the *setBounds()* method is called within the *outOfBoundsDetect()* method. Through the use of the *panda3d.core* library, the entire robot is given x, y, z coordinates associated with its perimeter. These bounds are visually represented using *LineSegs* as a separate node, then reparenting this to the root node of the Robot. These *LineSegs* are initially hidden, however are displayed to the user when they click on the Robot object after the scene has been rendered. The *outOfBoundsDetect()* method instantiates boundary boxes around each Robot object and produces an error message if any of these boundaries fall beyond the scope of the configuration.

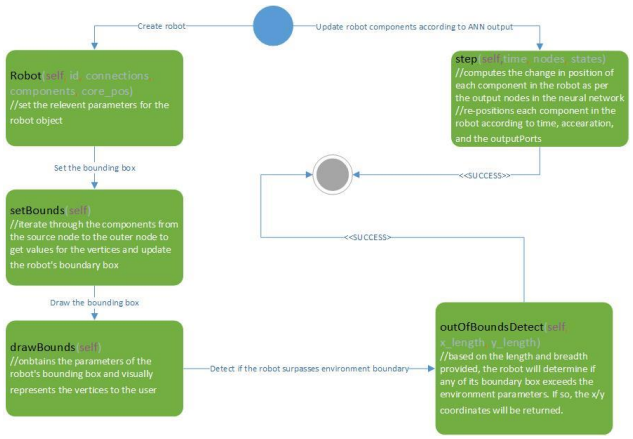


Figure 18 *Overview of Robot Algorithms*

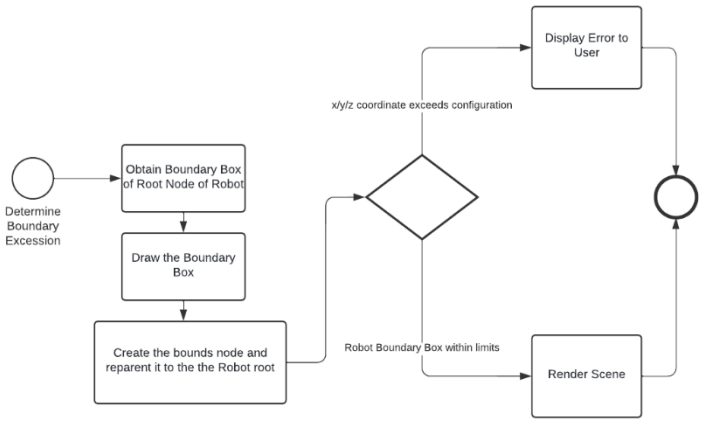


Figure 19 *outOfBoundsDetect() and drawBounds() Algorithm*

ii. Stepping the Robot

Upon retrieving the output of the ANN, the *step*() function is fed the time, output nodes of the ANN and an array consisting of the data produced by those output nodes. Based on this information, the acceleration of each component is calculated using the component's weight and gravity (9.8 m/s^2). From this, Equation 1 is used to calculate the change in position of each component within the Robot.

Bricks are set to have masses heavier than Hinges, and thus their change in position will be slightly greater. Additionally, as the effect of the mechanical oscillations will undergo damping as time passes, the change in position of each component that is not an output node is scaled down by a factor of 10% as the effects work towards the core of the robot. An overview of this algorithm can be visually represented by Figure 20.

$$\Delta x = \frac{1}{2} a t^2$$

Equation 1 *Calculation of Delta X*

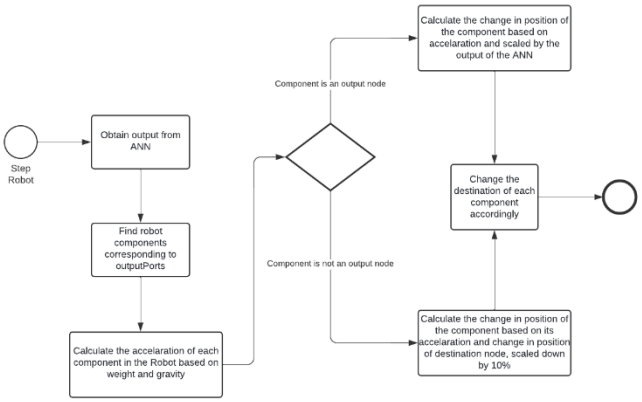


Figure 20 *step() Algorithm*

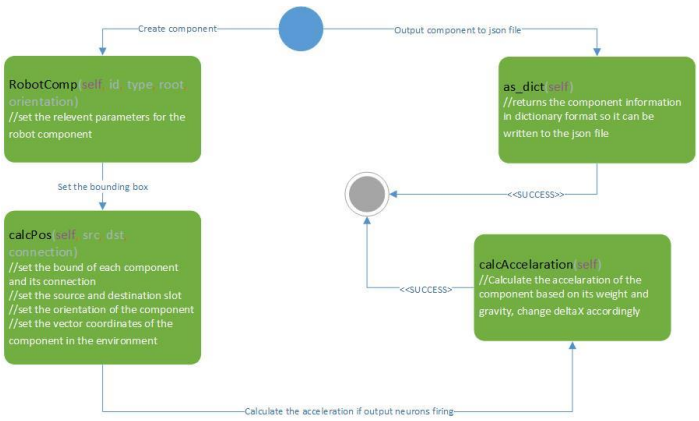


Figure 21 *Overview of RobotComp Algorithms*

4.4.5 RobotComp

This class defines the basic parameters of the components that make up a Robot. It is used extensively by RobotGUI during the build process. Additionally, RobotUtils makes use of this class a considerable amount when creating new robots and writing them to files.

As the Robot class and Connection class depend on having RobotComp objects to be instantiated, these objects are key building blocks to the overall system. Moreover, sensor nodes feed the neural network, and motors output the response from the network. An overview of these algorithms is represented in Figure 21.

The most important function of this class is the calculation of the individual positions of each component. This algorithm is represented in-depth by Figurre 7.2. A key library used in the position of each component is *panda3d.core* where the LVector2f data structure is vital to the definition of <x,y,z> coordinated of each component.

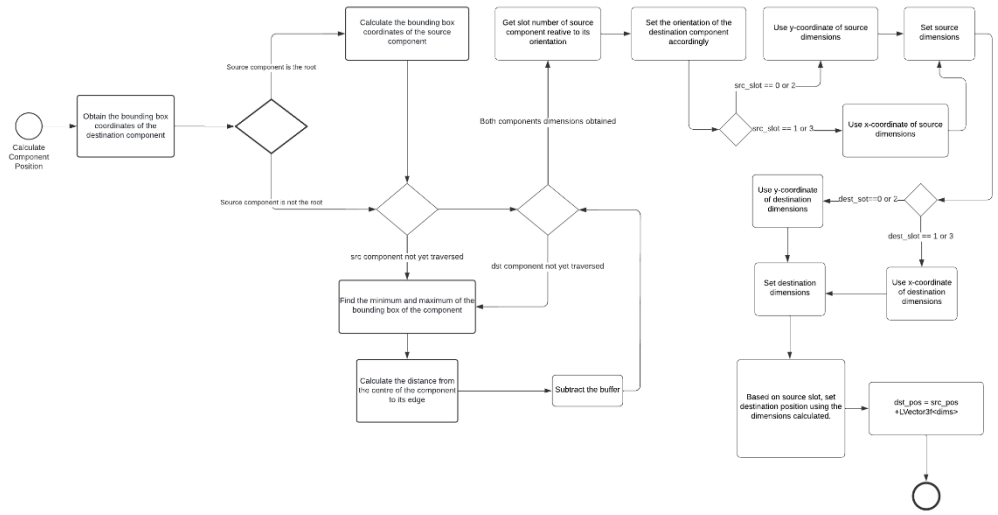


Figure 21 *CalcPos() Algorithm*

4.4.6 Connection

The most noteworthy method in the Connection class is the *standardiseSlots()* Algorithm. This function takes the slot system that RoboGen uses and then converts it to a more intuitive system which is then used by the RobotComp *calcPos()* method. A diagrammatic comparison can be found in Figure 22.

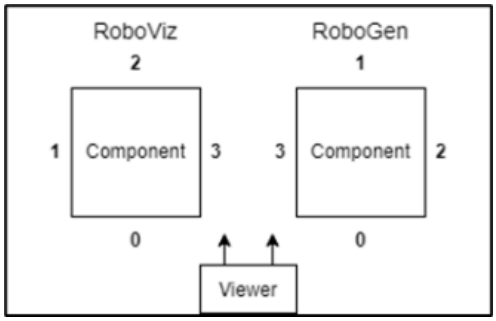


Figure 22 *Robogen and RoboViz between slot system comparison*

The RoboGen slot numbers were seen as confusing by the project team, and so a decision was made to adopt a more intuitive system and simply include this 'adapter' method so that one could easily switch between the two if needed. The comparison between the two methods can be identified in Table 2.

Table 2 Comparison between Slot Number Theorems

Slot Number	RoboGen	RoboViz
0	front (side closest to viewer)	front (side closest to viewer)
1	back (furthest from viewer)	left (side to left of viewer)
2	right (side to right of viewer)	back (furthest from viewer)
3	left (side to left of viewer)	right (side to right of viewer)

In other words, slots are labelled clockwise 0 to 1 starting from the side closest to the viewer. This method is called from the Environment *renderRobot()* method.

4.4.7 ANN

The ANN class holds the functionality of a singular robot's brain. It is initiated in the *createBrain*() method in RobotUtils upon the processing of a homogenous, lone robot. This neural network has been constructed to obtain data from light sensors within the Robot's framework and *feed()* this data into its input neurons, based in the core of the Robot. Aside from input nodes, there are additionally Sigmoid and Oscillator nodes. Sigmoid nodes can be calibrated with a gain and bias. Oscillator nodes, however, consider a period and phase offset. These variables change in direct proportion to the amount of time the neural network is propagating.

The states of the output nodes are collected after the *step()* function, represented by Figure 23. This function sets the states of each of the nodes, which are then retrieved by calling *fetch().* An overview of the methods included in this class can be seen in Figure 24.

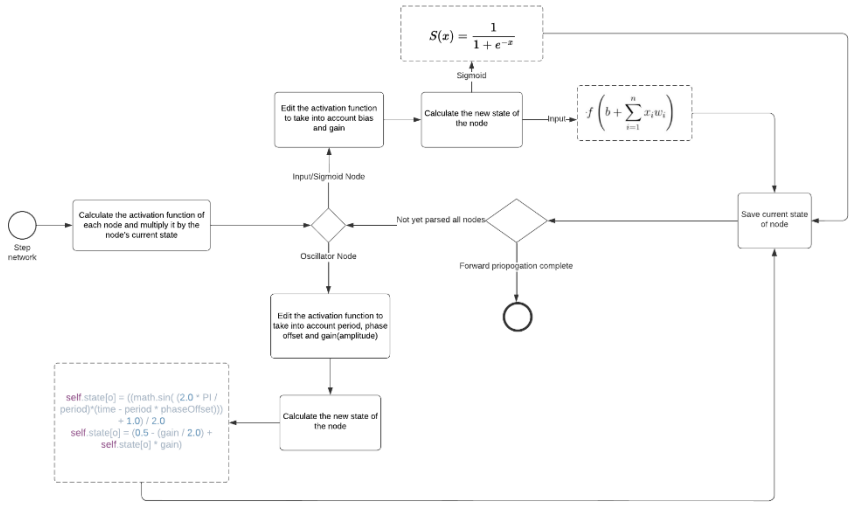
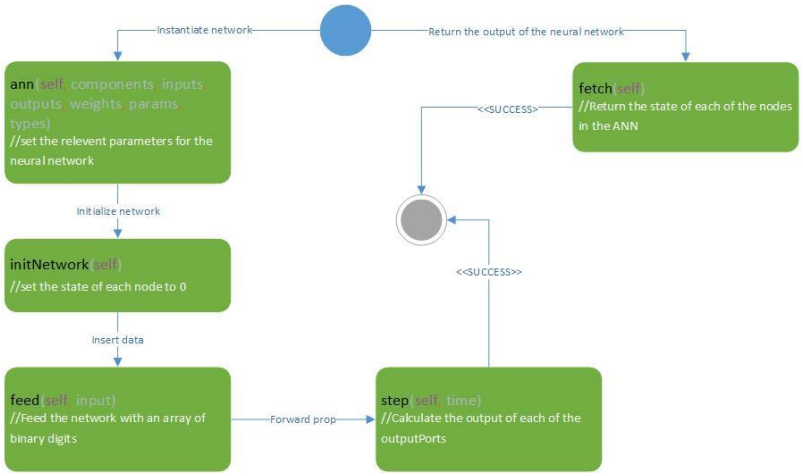


Figure 23 *step() Algorithm*

Figure 24 *Overview of ANN Algorithms*



5. PROGRAM VALIDATION AND VERIFICATION

5.1 Quality Management Plan

Table 3 *Summary of Testing Plan*

Process	Technique
Class (Unit) Testing: Test methods and basic functionality of classes	Partition and guideline testing
Regression Testing: Test if existing functionality is broken by new functionality	Selective and progressive testing
Integration Testing: Test if the interaction between new and existing classes breaks either's functionality	Black-box (behaviour) testing
System Testing: Test how the system performs with all its components put together	Stress and recovery testing
Validation Testing: Test whether customer requirements are satisfied	Use-case testing

5.1.1 Class and Unit Testing

Individual group members performed Class and Unit testing on their own code as they were developing new features. This was done throughout the development process at the level of methods or basic code functionality.

This was effective in rooting out issues in the code before anything was integrated as it provided fine-grain validation of functionality.

The following methods were used:

1. Partition testing: possible inputs were chosen from a range of different values, to get good coverage of data
2. Guideline testing: inputs were chosen with the knowledge of known failure points in the software.

5.1.2 Regression Testing

Regression testing was performed when a team member 'finished' adding a chunk of new functionality. Automatic tests for existing code using or being used by the new functionality would be rerun to determine if any existing functionality was broken by the new features and occasionally, if a requirement changed, existing tests would have to be rewritten to fit the new requirement.

This was effective in ensuring that each new version was only adding functionality to the main build and not causing any issues with the existing functionality.

The following methods were used:

1. Selective testing: a subset of existing tests for existing functionality that uses or is used by the new functionality were run
2. Progressive testing: newly designed or rewritten tests were run for when a requirement or existing functionality changed.

5.1.3 Integration Testing

Integration testing was performed when a new class was added to the main build. This would be whenever a team member deemed that class to be 'done' (i.e., sufficiently tested and bug-free). Tests were run on the new class and any classes that interacted with it to determine if any existing class's or the new class's functionality was broken by the integration.

This was effective in ensuring that the functionality of the individual classes involved in the integration wasn't affected.

The method of black-box (behaviour) testing was used. Only the external behaviour of the software was considered when choosing test data

5.1.4 System Testing

System testing was performed in the later stages of development. The aim was to attempt to take the software to its limits, both to quantify them and to highlight areas of the code that needed optimisation.

This was effective in guiding design decisions with regards to performance-functionality trade-offs as new functionality often came at the cost of higher overhead and a corresponding decrease in performance.

The following methods were used:

1. Stress testing: the software was tested with an extreme number of robots and the FPS (Frames Per Second) was monitored as a performance metric.
2. Recovery testing: the software was purposefully given erroneous input and its ability to recover and to respond to this incorrect input was monitored.

5.1.5 Validation Testing

Validation testing was performed after the tentative 'final' build was created. The aim was to ensure that all final customer requirements were met by the software. Where the software only partially met a requirement, a suggestion for a change was made and, after all team members agreed it would resolve the requirement, it was implemented in the code.

This was effective in ensuring that the software was able to completely fulfill the customer's and, by implication, the user's requirements. The method of use-case testing was used, in which each team member worked through the list of use-cases and attempted to use the software to complete these.

5.1.6 Testing Suites

Two testing suites were used to test the software at various stages of development.

i. *test_activation.py*

This suite tests the initialisation of each object class and some small functions. It was written using unittest. Its aim is to validate that all objects are being initialised properly, especially with composition and aggregation relationships such as between Connection and RobotComp, and Robot and Connection.

Table 4 *Summary of activation tests carried out*

Test	Data Set	Test Cases
		Normal Functioning
RobotGUI	Constructor arguments	Passed
RobotUtils	Constructor arguments	Passed
Environment (*toggleLabels*)	Constructor arguments	Passed
RobotComp (*calcAcceleration*)	Constructor arguments	Passed
Connection	Constructor arguments	Passed
Robot	Constructor arguments	Passed
Brain (*feed, step*)	Constructor arguments	Passed

ii. *test_,method.py*

This suite tests the main function of the program including helper methods, error handling, out of bounds and collision detection, file operations, and robot rendering. It was written using *pytest*. It contains numerous visual tests that allow the tester to ensure that the display of different robot configurations is correct. To verify that the robots were rendering correctly, RoboGen was used to render them and then compared to the RoboViz robots.

The purpose was to validate the output of the following methods and ensure all program functions were operating correctly after integrations or new code, but also to ensure that, once written, methods were functioning as prescribed.

Table 5 *Summary of method tests carried out*

Test	Data Set	Test Cases		
		Normal Functioning	*Extreme boundary cases*	*Invalid Data*
Helper methods (*standardiseSlots*, *calcPos*)	*Connection* objects with different components and slot numbers	Passed	N/A	Pass (invalid slot error handled)
RobotGUI error handling (*runSim*)	Various incorrect combinations of configuration, positions, and JSON files	Passed	N/A	Passed
Out of bounds & collision detection (*outOfBoundsDetect*, *collisionDetect*)	3 sets of Robot bounds (normal, edge, out/collision)	Passed (*bounds_normal*)	Passed (*bounds_edge*)	Passed (*bounds_out/collision*)
RobotUtils file operations (*configParse*, *posParse*, *writeRobot*)	Set of 3 correct configuration, positions, and robot file paths	Passed	Passed	Passed

5.2 Discussion of Test Results

5.2.1 Stress Testing

Stress testing was performed by using large amounts of robots to render with. We used 1000 robots in the .JSON file along with an environment and position file that worked with the 1000 robots.

The result of trying to render this many robots will still allow the program to run normally but at a slower speed. Rendering time is a lot longer than with rendering less robots but it takes less than 2 or 3 minutes on average. Once it is rendered looking at all the robots makes Panda3D run at about 3 fps

but zooming in on a singular robot allows Panda3d to run at between 30 to 60 fps depending on the system you use. The use of extreme and edge case data uncovered potential fault points within the program and allowed the team to implement the required error handling.

5.2.2 Validation Testing

User requirements to test: inputting 3 specific files in CLI, zooming in and out, panning, inputting 3 specific files in GUI, renders robots correctly.

Table 6 *Validation Testing Results*

INPUT	OUTPUT
position, configuration and robot files into CLI	robots specified in the files are rendered correctly and in the correct positions
position, configuration and robot files into GUI	robots specified in the files are rendered correctly and in the correct positions
once robots are rendered user right clicks and moves their mouse forward or back	environment user interface reacts by zooming in and out
once robots are rendered user left clicks and moves their mouse around	environment user interface reacts by panning in the direction the user drags their mouse

Once the robots were rendered, they were compared to the online Robo-Gen version to make sure there were no differences, in which case they are not from the tested files

5.2.3 Automatic Testing

Through the use of automatic tests, the team was able to work swiftly and efficiently whilst remaining confident in the fact that new contributions to the code did not break pre-existing functionality. This meant that the program's correctness could be verified throughout the development process, with new tests being added with new functionality.

6. WORK DISTRIBUTION

6.1 Fynn Young

Fynn always kept the codebase in an error-free state, performing all integration and regression testing throughout the project. He was responsible for the merging of each team members code with the working version on his side. He employed all the functionality of the Environment class, including the rendering of robots and their components, and the calculations that went with this endeavour. In addition to this, he implemented the robot BUILD function, including the writing of Robot objects back to JSON files and error detection that accompanied incorrect instantiation. He also executed the AUTO-PACK function and HELP button on the GUI. He performed extensive error checking, including the detection of robots being rendered atop each other (and therefore found and implemented the rectangle-packer library) and out of bounds detection. He realized the test_method.py class method tests and wrote documentation to accompany this.

6.2 Daanyaal Gamieldien

Daanyaal was in charge of keeping the team sticking to the scope of the project. His initial role in coding was the implementation of inheritance amongst classes. Additionally, he determined and implemented the GUI library suitable enough for user inputs for file browsing. He handled the GUI's error handling for the input files from the user. Daanyaal also performed many validation and stress tests in order to ensure the program was robust. He constructed the option for either Command Line Interface or GUI input. The *LastRender* function in which the user may save their previously rendered scene was implemented by Daanyaal.

6.3 Claire Fielden

Claire was responsible for ensuring meetings were held constantly and products were delivered timeously and in a fit state. She implemented the code to parse the input files from the user and created the basic structure of the Robot, robotComp and Connection classes to store this information. When the GUI was erected, she ensured the command line interface still took user input correctly and handled the logic and format checking, as well as delivery of error messages. She worked extensively with Fynn to try and fix the orientation errors causing the rendered robot not to match the one on RoboGen, however this ended up being a hard-coded nested if{} loop that Fynn quickly replaced with a one-liner. Claire also implemented the brain and all attached functionalities.

Everyone contributed to unit tests in their own capacity, as well as overall debugging and basic fixes within the codebase.

7. CONCLUSION

The aim of this project was to develop a tool to visualise a robot swarm in its task environment. This required the acceptance of numerous file formats containing the definition of a reference robot, whether it be a heterogenous or homogenous swarm. For research purposes, the user requested unmarred analysis of the swarm's collective gathering in a 3D-bounded area. This intention was accomplished through the efficient creation and connection of models as specified by the user's input files. The rendered environment allows numerous functionalities for the analysis of the rendered swarm, as well as the ability to auto-pack the robots, detect collision and build one's own robot. The neural network comprising the brain of the robot has also been constructed for further development. For ease of use, a Graphical User Interface is available upon request.

Therefore, the initial objective has been achieved through the creation of a well-structured application that implements all elements of object-oriented programming techniques. This system is unduly robust, not only in its performance and ability to handle excessive computational load but amplified by the endless amounts of tests that accompany the program. In addition to these factors, the code is well-documented to aid maintenance in the future, as well as assist users in the use of this application.

8. REFERENCES

9. Anon, (2022). Adam Optimizer PyTorch With Examples - Python Guides. [online] Available at: https://pythonguides.com/adam-optimizer-pytorch/ [Accessed 30 Apr. 2022].
10. Brownlee, J. (2019). A Gentle Introduction to Cross-Entropy for Machine Learning. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/cross-entropy-for-machine-learning/#:~:text=Cross%2Dentropy%20can%20be%20used [Accessed 30 Apr. 2022].
11. Brownlee, J. (2021). How to Choose an Activation Function for Deep Learning. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/#:~:text=for%20Output%20Layers-.
12. Data Science Stack Exchange. (n.d.). overfitting - Why not use more than 3 hidden layers for MNIST classification? [online] Available at: https://datascience.stackexchange.com/questions/22173/why-not-use-more-than-3-hidden-layers-for-mnist-classification [Accessed 30 Apr. 2022].
13. EDUCBA. (2022). PyTorch Sigmoid | What is PyTorch Sigmoid? | How to use? [online] Available at: https://www.educba.com/pytorch-sigmoid/.
14. IBM Cloud Education (2020). What are Neural Networks? [online] www.ibm.com. Available at: https://www.ibm.com/cloud/learn/neural-networks.
15. in (2010). How to choose the number of hidden layers and nodes in a feedforward neural network? [online] Cross Validated. Available at: https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw.
16. neptune.ai. (2020). PyTorch Loss Functions: The Ultimate Guide. [online] Available at: https://neptune.ai/blog/pytorch-loss-functions#:~:text=Broadly%20speaking%2C%20loss%20functions%20in [Accessed 30 Apr. 2022].

17. sentdex (2019). Training Convnet - Deep Learning and Neural Networks with Python and Pytorch p.6. YouTube. Available at: https://www.youtube.com/watch?v=1gQR24B3ISE [Accessed 20 Dec. 2021].
18. www.pluralsight.com. (n.d.). Building Your First PyTorch Solution | Pluralsight. [online] Available at: https://www.pluralsight.com/guides/building-your-first-pytorch-solution.
19. www.youtube.com. (2019). Building our Neural Network - Deep Learning and Neural Networks with Python and Pytorch p.3. [online] Available at: https://www.youtube.com/watch?v=ixathu7U-LQ&list=PLQVvvaa0QuDdeMyHEYc0gxFpYwHY2Qfdh&index=4 [Accessed 30 Apr. 2022].
20. www.youtube.com. (2021). Making Predictions with PyTorch Deep Learning Models. [online] Available at: https://www.youtube.com/watch?v=0Q5KTt2R5w4 [Accessed 30 Apr. 2022].
21. www.youtube.com. (2022). Making a Neural Network for MNIST in Pytorch | Pytorch Full Course Part 3. [online] Available at: https://www.youtube.com/watch?v=9Bxf9voEbMg [Accessed 30 Apr. 2022].
22. Andersson, D., 2017. Benchmarks — rectangle-packer 2.0.1 documentation. [online] rectangle-packer 2.0.1 documentation. Available at: <https://rectangle-packer.readthedocs.io/en/latest/benchmarks.html#time-complexity> [Accessed 20 September 2022].
23. Panda3D Manual. 2019. DirectGUI — Panda3D Manual. [online] Available at: <https://docs.panda3d.org/1.10/python/programming/gui/directgui/index> [Accessed 16 September 2022].
24. Panda3D Maunal. n.d. The Scene Graph — Panda3D Manual. [online] Available at: <https://docs.panda3d.org/1.10/python/programming/scene-graph/index> [Accessed 15 September 2022].
25. Rolland, R., n.d. TimeComplexity - Python Wiki. [online] Python Wiki. Available at: <https://wiki.python.org/moin/TimeComplexity> [Accessed 16 September 2022].
26. Brownlee, J. (2019). A Gentle Introduction to Cross-Entropy for Machine Learning. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/cross-entropy-for-machine-learning/#:~:text=Cross%2Dentropy%20can%20be%20used [Accessed 30 Apr. 2022].
27. Brownlee, J. (2021). How to Choose an Activation Function for Deep Learning. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/#:~:text=for%20Output%20Layers-.
28. IBM Cloud Education (2020). What are Neural Networks? [online] www.ibm.com. Available at: https://www.ibm.com/cloud/learn/neural-networks.
29. RoboGen. n.d. » Guidelines for writing a robot text file. [online] Available at: <https://robogen.org/docs/guidelines-for-writing-a-robot-text-file/> [Accessed 17 September 2022].