

Capstone: RoboViz

Final Report, September 2022



Authors:

Benjamin Chiddy

CHDBEN002

Jonty Doyle

DYLJON001

Hamza Amir

AMRHAM001

Contents

Introduction	3
Requirements Captured	3
Design Overview	6
Implementation Details	8
Program Validation & Verification	12
Conclusion	15
Appendix	16
User Manual (See README.md for markdown version)	23
Responsibility Breakdown	26
References	27

Abstract

This report details our implementation of the RoboViz application. In addition to the core requirements requested by the clients we saw it fit to expand our scope to include additional features that led to a piece of software that was usable at a high-level by both robotics researchers and people who are new to the field. To that end we ensured that our final solution was easy-to-use for both of these groups through the addition of a command line interface as well as a GUI. A large focus was also placed on performance to ensure the system could handle and interact with large scale swarms. To demonstrate this we conducted substantial stress testing on the system and found the upper limit for swarms was in the order of $10^4 + C$ (where C is some arbitrary constant). Ultimately, the project was successful in its goal to deliver a working product which delivers on the original requests but also goes over and above to provide an engaging experience for both experienced robotics researchers as well as newcomers to the field.

Introduction

The purpose of this project is to provide robotics researchers with a **local** 3D visualisation tool which will allow them to simulate robotic swarms without large investment into physical experiments.

Heiko Hamann states in his 2015 book, *Swarm Robotics: A Formal Approach* - that it is “useful” to run initial simulations before conducting any robotic swarm experiments as tasks such as charging, robot position and flashing become tedious for large numbers of robots (Heimann, 2015). This is in addition to hardware issues such as reliability and robustness when dealing with many robots that make jumping straight into physical experiments barely feasible. Our visualisation tool will seek to close this gap and provide researchers with a cost effective and robust way to collect experimental data for robotic swarms.

Secondary to this, a tool for visualising robotic swarms provides a compelling way for researchers to communicate their research with the layman (those without sufficient knowledge of robotics engineering). This could be used to secure investment for large-scale physical experiments once a proof of concept has been shown using the simulation tool. As a byproduct of this focus, researchers will also be able to use the visualisation tool to communicate their research with friends, family and possibly inspire young scientists to pursue a career in robotics through educational demonstrations.

We also note the work in this field by the RoboGen team who have produced a downloadable version of this tool which is already popular amongst robotics researchers. However, this tool is extremely large and takes time to install. RoboGen have also produced a lightweight web-based version of the tool (link in report references), however this version does not allow customisation of models offering only a small demo.

Our intention is to provide an approachable, lightweight and customisable tool which does not seek to replace the use of RoboGen but instead provide a stepping stone into the world of robotics visualisation.

Requirements Captured

Functional

Our functional requirements are split across two categories; firstly the core requirements which are the features requested directly by the clients and secondly the additional features added by our team. After completing the core features, the team ideated additional features and prioritised those which would make the highest impact for our users (outlined in use case section).

1. Core Requirements

ID	Requirement	Description
1	Parsing config files	System should parse .txt config files given to it via the command line.
2	Parsing robot files	System should parse JSON robot files given to it via the command line. These could include either single robot (homogeneous) descriptions as well as multiple (heterogeneous).
3	Parsing environment files	System should parse .txt environment files given to it via the command line
4	Rendering a 3D robot model	System should render a 3D robot model using config, robot and environment files.
5	Model navigation	Users should be able to interact with the model environment by zooming in and out or moving around.

2. Additional Features Added

ID	Requirement	Description
1	Environment Interaction	Users should be able to interact with the built environment, during runtime.
2	Ability to save and load models	Users should be able to save their custom models and then load them again at a later stage. Even after the application has been closed and reopened.
3	Rendering > 100 robots	System should be able to handle large models (>100 robots) without crashing.
4	Focusing/ Defocusing on a specific robot.	Users should be able to focus on one specific robot when looking at a render of an entire swarm.
5	View information about render.	Users should see information about the success of their current render printed to the screen.

Non-Functional & Usability

Because the core requirements of the product were regarding the rendering of 3D graphical models a large focus was placed on the performance of the system and its ability to efficiently handle various loads. This forms a major part of our software testing discussed in an upcoming section. In addition to this, the secondary focus of our product was its use by people not necessarily familiar with robotics research - for this reason a lot of

thinking went into making the product as intuitive and easy to use as possible. Following on from this, because the system relies very heavily on user input - attention was given to making the I/O flow as intuitive and robust as possible to prevent any runtime crashes. Finally, as discussed in the introduction our intention was for this product to serve as a lightweight version of the RoboGen platform - for this reason focus was placed on making the system runnable on various machines.

ID	Requirement	Description
1	Rendering performance	System should be able to render large multiples of robots without significant strain.
2	Ease of use	System should be easy to use for both technical and non-technical users.
3	Robust I/O	System should be able to handle errors in the files passed to it.
4	Portability	System should be able to run on any system that has python installed.

Out of Scope (Possible Future Releases)

With the requirements of this project being very specific, we thought it would be appropriate to summarise features which fall outside of the scope of this version of the product. These features could possibly form part of future releases.

ID	Requirement	Description
1	Moving Robots	Robots rendered in the model can move around the environment and interact with each other.

Use Case Diagram

As discussed in the introduction, the main focus of our visualisation tool will be providing robotics researchers with a robust tool to visualise robotic swarms. Secondary to this we have also decided to cater for users that are not familiar with robotic swarms and therefore will be using the tool either for education or entertainment. Because of this it was important that we consider use cases which promote ease of use and accessibility. The use case diagram can be seen in the appendix as **Fig.1**.

Use Case Narratives

The initial request from the client was to create a tool that would be executed via the command line, however we feel that this is cumbersome and prone to errors - especially with our inclusion of a group of secondary users. This has inspired us to create a Graphical User Interface (GUI) which will allow users to save and load models. The GUI will take the form of a main menu which will offer users the option to run a custom model, load a pre-saved model or exit the system altogether. These narratives can be found in the appendix, directly after the use case diagram (**Fig. 1**)

Design Overview

Physical Architecture

The source files are organised into python *modules*, with each module serving a distinct role. This design choice results in the limitation of coupling *between* modules. Below is a breakdown of this physical structure.

Structure of Source Files

src/	Stores the Core module, holding core classes of Robot , Component and ComponentTree , among others. This includes the entry point of the application.
src/app/	Location of the App module, holding the Environment class as well as other (highly coupled) classes within.
src/app/ui	Location of the UI module. All classes and methods of pertaining to the User Interface are stored here (primarily the Console)

Logical Architecture

Separation of concerns was a high priority in the logical design of our implementation, further aiding in minimising coupling. The logical architecture is primarily segmented by the **App** module, which inherits/depends on Panda3D, while other classes in the **Core** module (the **src/** folder) do not. See **Fig. 2**, containing a (simplified) class diagram representing the logical architecture (note that this diagram excludes UI related classes), the following brief description refers to the aforementioned diagram:

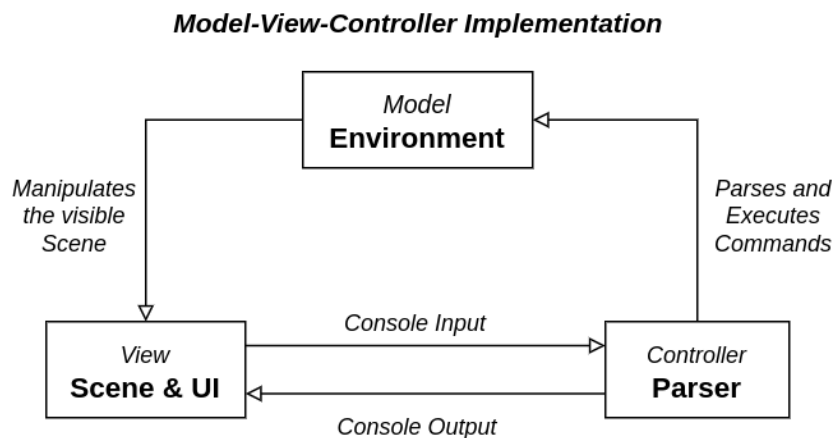
The software's entry point is through the command line, handled by **Main**, which is responsible for handling arguments and the instantiation of the core **App** module. Depending on the arguments, the **App** class will initialise both the **RobotData** and **EnvironmentConfig** objects, for details on the (purpose of these objects see implementation below). The **App** class *inherits* from Panda3D's **ShowBase** class which affords it to interface with the functionality of Panda3D. The **Environment** class is the central point of the application (see

implementation details regarding its contents), interacting with all classes aiding in rendering and presenting an attractive 3D Environment to the user.

The primary interaction point (see the architectural pattern for further details below) is through the **Parser** class. This class is responsible for populating and parsing commands during runtime by which the user can interface with the environment (this process is purely dynamic, populating commands via the public methods of the **Environment** class - see implementation details below).

Externally to the **App** module, the **Robot** class represents the logical abstraction of a Robogen robot, and its loading/rendering functionality is handled through the **RobotModel** class in the **App** module. A **Robot** consists of an N-Ary tree of **Component** objects, by which the **ComponentTree** object serves as a helper class to traverse the tree and find particular nodes. Each specific component type (**CoreComponent**, **IrSensor**, etc.) *inherits* from the base component class, providing specific attributes required for rendering and placing the components in the scene.

Architectural Patterns



A Model-View-Controller (MVC) architecture was chosen for our implementation, this design choice was justified by the necessity for direct interaction and manipulation of loaded *Environments* (see implementation for further details and definition of an *Environment*). The above diagram depicts the flow of interaction using this pattern. The *View* consists of the 3D-Scene and the **UI** module (primarily the **Console** class), the **Parser** (see implementation) is the interface to the **Environment**, which represents the *model*. **Console** input & output provide the user direct feedback, and the public methods stipulated in the **Environment** specify the scope of control by which the parser can manipulate it.

Notes on Implementation Efficiency & Algorithms

Given the problem set, no notable core (well-defined) algorithms were required to be implemented (the process of placing robots, however, is described below). On a smaller scale, Python dictionaries & sets were preferred to lists when possible due to their hash-like capabilities. Dynamic, general solutions were also preferred over specific ones, which is exemplified in the building process - as this contains almost no conditional (hard-coded) logic and operates in a dynamic fashion with respect to loaded model sizes. Examples of elegant & efficient solutions are using Spherical Coordinates to handle the camera orientation & populating/parsing commands sent by the user. Algorithmic complexity was a concern, however performance bottlenecks stemmed from polygon counts & geometry handling, in which solutions were found (see below).

Implementation Details

The software was written in Python (version 3.5+), with a single runtime dependency: **Panda3d**. Development and testing was conducted in a Linux & Unix (MacOS) environment. Version control was used extensively (see *git log* for further indication). Consistent code convention (snake case) and formatting (PEP-8) was adhered to. See the User Manual for further information regarding the directory structure.

For development, distribution and testing: virtual environments are handled through the **venv** module, and distribution through **pip** and **setuptools**. Makefiles are used to automate and simplify this process. The codebase is heavily documented, for further details on classes not covered here please refer to the source code *docstrings*.

Repository Structure & Model Pipeline

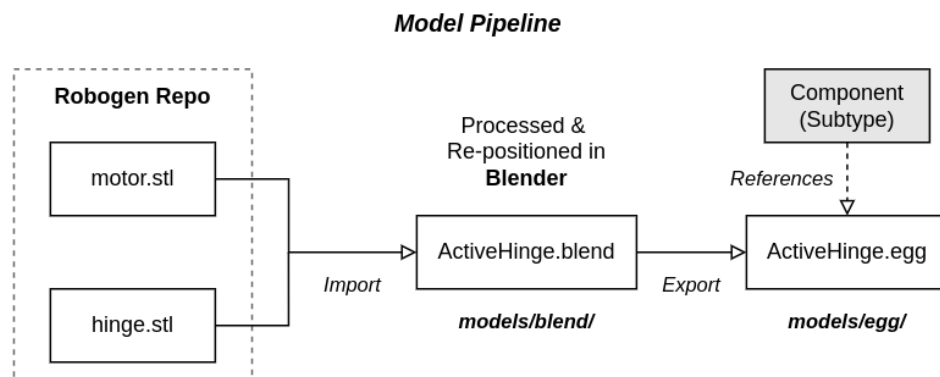
Repository Breakdown

config/	Holds sample configuration (and position) files according to the expected input.
data/	Contains sample input data (output data from <code>robogen-evolver</code>), with sample minimal robots used to visually test placement logic.
docs/	General documents used in development (including diagrams)
assets/	Model and environment data files. Various project files used in model development.
src/	Python module root, housing the architecture (broken down further below)
venv/	Python virtual environment, present after install.
makefile	~

README.txt	Contains installation & manual
LICENSE	GNU GPL v3 License.
requirements.txt	Dependencies (panda3d & setuptools)
setup.py & pyproject.toml	Build specifications & installation script, used by <code>setuptools</code> ,

A strict and logical directory structure was maintained throughout the development process, the above serves as a reference to all the components in the repository.

Model Pipeline



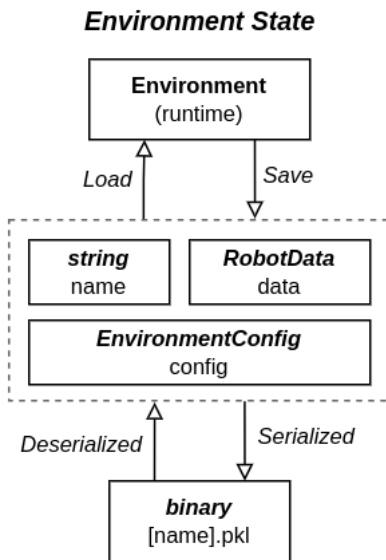
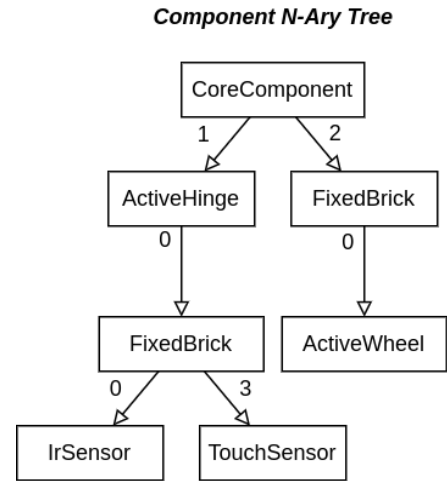
As the development process involved rendering models in a 3-D environment, a model pipeline was required to be established. The above shows the construction of the ActiveHinge model, and the process/pipeline used in development. **Blender** was used as the modelling tool, importing Robogen's models intended for -3D printing (.stl), which are processed and exported to panda3D recognisable .egg files. The Component Subtype class then holds a file-path reference to this file, which is subsequently loaded and rendered during the build process (see below). The benefits from such a process, to name a few are:

- Easily transform (rotate, move, scale) improperly placed components during development
- Increase performance by decreasing polygon count during processing

Notable Data Structures

Robot Components

Each Robot internally holds an N-Array Tree consisting of Component Subtype objects. A tree was chosen due to the inherent parent-child relationships found in Robogen's robot specification platform. In particular, the fact that each component's position (and orientation) in space depends on its parent component, which in turn depends on *its* parent component, and so on. Additionally and conveniently, objects in a 3D environment are often placed using a Scene Graph [see [here](#)] in a similarly hierarchical manner, with Panda3D doing the same. Being a tree, each **Component** holds a dictionary of child components (with the key being the slot in which it is housed). The adjacent diagram (representing a fictional component) best describes this, with each specific Component (inheriting from the component base class) having their children indicated with arrows and their respective slot by number.



The Environment

The Environment data structure (contained in `environment.py`) holds all runtime-state regarding an *Environment*. An *Environment* is defined loosely in the Functional Requirements, though implemented strictly as a 3-tuple with a **name**, some robot **data** (of type `RobotData`) and **config** (of type `EnvironmentConfig`).

Note that this does *not* contain any data relating to Panda3D, nor any data regarding rendering/geometry. This was an active implementation constraint such as to minimise saved/loaded binary data sizes. The environment can be reconstructed given this 3-tuple and thus saved & loaded during runtime, as described by the adjacent diagram. Furthermore such a strict definition of an environment allowed us to limit scope, and maximise functionality given such.

Rendering Implementation Details

Rendering the Environment

The most notable interaction between classes occurs when an environment - and its robot(s) - are being rendered. The rendering process is invoked through the two (and only) private methods in the **Environment**

class. These two methods, and their invocation, are discussed below (further consult Fig.4 - Sequence Diagram in the appendix):

- **build(name, self, config)**: This method is analogous to the outer loop in Fig. 4, and effectively primes the environment for rendering by parsing & validating the relevant configuration & data files. If valid, the terrain is added (per specification), **Robot** objects are created given the configuration and then subsequently passed to the **add-robot** method. Finally, render times for profiling are initialised.
- **add-robot(robot)**: Robots are added to the scene through the invocation of a **RobotModel** object (parameterised with a **Robot**), which is tightly coupled with Panda3D and responsible for holding all information regarding model loading, placement and collision detection. It is here that collision detection errors are queried and logged. Rendering is handled through this class, and discussed in detail below.

Rendering a Robot

As mentioned, the process of rendering a robot is carried through in the **RobotModel** class. The sequence of events involved in placing a set of components is loosely described below:

1. A root node in the Scene Graph is created, with the name of the robot ID. All robot components are then enumerated and iterated over. Using a tree traversal guaranteeing that no child node is specified before its parent.
2. For each component in the robot:
 - a. Fetch the model and orientate it as specified.
 - b. Create the respective slots by which the component connects to its parent.
 - c. Reparent this component's to its parent component's in the Scene Graph, thus orientating it relative to its parent.
 - d. Reposition the location of the newly placed component such that it sits flush to its parent. (computed of the child & parent components' model size).
 - e. If they slot-in, offset this placement by a fixed offset amount to provide the impression of connected components.
3. Return the root node to the environment, and reparent this to the Render Node (if there are no collisions with existing robots), thus rendering the robot.

Interacting with the Environment

The interface implementation was intentionally minimal as the software is intended to be used as a tool fit for a specific purpose, with visible elements including the Command Line (and its output) and an Input Box for naming/saving environments. The application thus has two modes: Input Mode, and Command Mode. All UI related source code is contained within the **app/ui/** directory. The Console implementation is discussed in detail below.

Console & Command Parser

Two files pertaining to the implementation of the console, `console.py` and `parser.py` represent the User Interface (the physical input box etc.) and the logical command interpretation respectively. The command interpretation is fully dynamic, populating commands pulled from the public methods of the **Environment** class, and further sourcing usage and help from the methods' docstrings. The result is minimal coupling between the environment and the parser, allowing for easily adding new commands (by just adding new public environment methods, which mutate the environment and return the string printed to the console). The **Parser** further includes an *Autocomplete* feature, allowing the user to enter Tab to autocomplete commands and parameters to certain commands.

Furthermore, a **Logger** was created to help log events for profiling (see validation below) and provide the user with information regarding collisions, placed robots and out of bounds errors etc.

Program Validation & Verification

Approach to testing

As mentioned many times earlier our vision for the product was to provide robotics researchers with a robust tool for visualising robotic swarms. With this being a graphically intensive task our testing was heavily focussed on the system's ability to handle intense loads. For this reason we chose to stress test the system, by rendering many multiples of robots. In addition to this, we also focussed on usability testing which gave us insight into how those not familiar with robotics research might use the system. Finally, we used validation testing to ensure that all core requirements (requested by the clients) were met. This was done in a "*big bang*" fashion at the very end of every sprint to ensure all core features remained working no matter what functionality was changed or added.

Summary Testing Plan

Process	Technique
Validation Testing	Acceptance tested based on functional requirements
System Testing	Stress testing the model performance when rendering models with multiple robots.

Functional Requirement Tests

1. Validation Testing

Below are the test cases we outlined in our prototype doc as what would fall part of our eventual user acceptance tests. For this reason we opted to test these features in a big bang fashion - at the end of our product development. Because of the importance of these tests we have included them in the main part of this report, whereas the results of the stress tests have been moved to the appendix.

Functional Requirement Test Cases

ID	Test Case	Input	Expected Output	Status
1.1	RoboViz Program handles Input, and error checking.	The program is run from the command line, included in these test cases are corrupt data, incorrect paths etc.	The user is notified if any corruptions or empty paths exist, and prompted.	Pass
1.2	Internal Robot Component Structure accurately reflects the output produced by Robogen.	Robogen's output JSON file [run through Robogen's parse_json.py script] and the output of Component Tree produced internally.	The two are identical, i.e the our internal representation of the robot matches Robogen's	Pass
1.3	Robot components are placed correctly in the 3D environment.	The output (3D environment) from our software, compared to the output of Robogen's visualisation software.	They are visually similar, with components placed and oriented in the same fashion.	Pass
1.4	Robots are placed in the correct locations given the position file input, handling robots that may overlap.	A position file is given which may not have identical coordinates, though results in the robots overlapping due to their structure.	The software is able to detect such errors, and notifies the user of such a result.	Pass
1.5	The model can be run with Multiple Heterogeneous Robots	Multiple robots are provided in the description file, of a different type.	3D Environment renders with 3 robots of different types placed within it, validated at the correct locations.	Pass

Non-Functional Requirement Tests

2. Stress Testing




We conducted stress testing to determine the systems performance running models that require large numbers of robots to be rendered, taking note of the loading time. To determine cases we used powers of 10 to assess the performance of the system across similar bands requirements. Note that we have split the testing for the homogenous robot case from the heterogeneous robot case as the latter requires n different robots to be rendered. Below we chart the average loading time, over 10 runs, of the software on two different systems - each with differing hardware facilities (graphics cards).

Homogenous Robots (Cart)

ID	Amount of Robots	Intel Iris (1.5GB) Performance (seconds)	AMD Radeon (4GB) Performance (seconds)
2.1.1	1	0.039	0.095
2.1.3	100	1.156	0.746
2.1.4	900	10.741	6.714
2.1.5	10,000	195.005	114.757
2.1.6	90,000	-	-

Heterogeneous Robots (n=3)

ID	Amount of Robots	Intel Iris (1.5GB) Avg. Performance (seconds)	AMD Radeon (4GB) Avg. Performance (seconds)
2.2.1	4	0.089	0.056
2.2.3	100	2.013	1.291
2.2.4	900	18.179	11.678
2.2.5	10,000	282.188	173.889
2.2.6	90,000	-	-

 Rendered & Usable	 Rendered but Unusable	 Unable to render
---	--	--

Conclusion

In the beginning of this report we outlined our intention to create an easy-to-use, lightweight and robust application that might provide a stepping stone to people interested in robotics research. In fact our goal was two-fold; we wanted our application to be useful and robust enough to be used by robotics researchers planning large scale experiments as well as approachable enough for those not familiar with robotics research at all. This was a tight line to walk, but we believe was ultimately achieved through meeting the core functional requirements early in the development process which allowed us to use remaining sprints to focus on additional features that would improve the performance and usability of our final solution.

Through the development of a quick and easy to manipulate command-line-interface we hope to have improved the speed and efficiency of a researcher's workflow. To this end, we also placed a lot of focus on the performance of the application and its ability to handle large scale swarms of robots - the kind that needs to be implemented in software before its realisation in the real world. To show this we conducted substantial stress testing against large scale swarms the upper limit of which seemed to be in the order of $10^4 + C$ (where C is some arbitrary constant) for machines with reasonable graphics cards. The GUI commands **focus** and **unfocus** are of particular importance as well, as they allow researchers to drill down on specific robots during large scale experiments - either debugging or analysis purposes.

The GUI implementation seeks to improve the usability of the application to a point where the system is easily usable for those not familiar with robotics research. All the functionality that's offered to researchers through the command line interface are available through the GUI console with easy to type commands, together with autocompletion for increased ease-of-use. The idea here is that it offers a less intimidating way to interact with the application, while still being able to achieve a lot. Being able to **save** and **load** are big draw-cards here, as it allows users to run default models without having to input any files. This allows new users to get up to speed extremely quickly and hopefully inspires them to start building their own models.

All in all, our RoboViz implementation meets the core requirements and goes over and above to deliver an engaging experience for both robotics researchers and newcomers to the field. Because of this we see our final solution as a great success and hope you will enjoy using it as much as we did making it!

Appendix

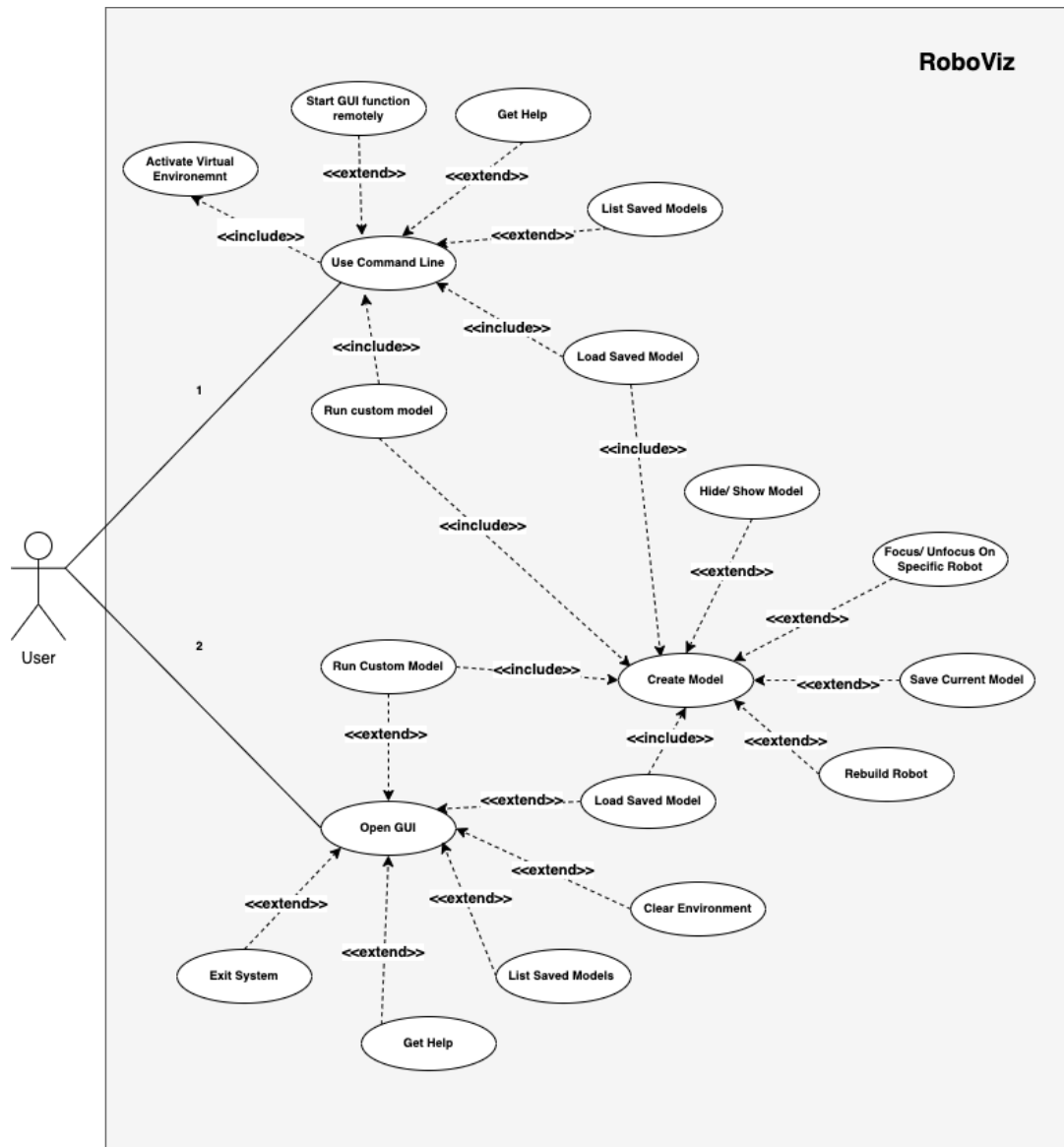


Fig. 1 Use case Diagram

Use Case Narratives

Below are more in-depth descriptions of our use cases. Note that these are high-level. For specific information on how to use the application please consult the user manual.

Action	1. Use Command Line	Priority: High
Main	<p>User firstly opens the command line in the RoboViz directory. They then activate the virtual environment (see user manual) which gives them access to the RoboViz command line tools.</p> <p>They can now simply open the GUI with the command <code>roboviz</code> or use the flag + command combination of roboviz -h to have a list of command line tools printed to the terminal.</p> <p>To list the available saved models they can use the command <code>roboviz -l</code> to have them printed to screen and use roboviz -L [model-name] to load a specific one.</p> <p>To run a custom model they use the <code>roboviz</code> command followed by the necessary config, positions and robot files in the following way; roboviz -c [config] -p [positions] -d [robot]. Note that these can be written in any order, as long as they are preceded by the correct flag (-c, -p or -d).</p>	
Alternative 1	If a user does not activate the virtual environment and tries to use the <code>roboviz</code> command - the terminal will respond with “ <i>command does not exist</i> ” (or something similar depending on your terminal). They can then either activate the terminal or simply run the command <code>make run</code> to start the GUI.	
Alternative 2	If a user gives incorrect file paths, a file does not exist or the files given are not consistent, the GUI will load with an error message telling the user what went wrong. They can then either close the application or retry running the model using the GUI console.	
Alternative 3	If a user tries to load a saved model that does not exist, the GUI will load with an error message telling the user what went wrong. They can then either close the application or retry running the model using the GUI console.	

Action	2. Open GUI	Priority: High
Main	<p>User opens the RovoViz directory and uses the command make run to open the RoboViz GUI. They will then see an empty environment screen and a welcome message. The user then presses the i key (inspired by VIM) to open the environment console and start interacting with the application.</p> <p>They can then enter the command help to list the available console commands relating to opening, loading, listing a model. To run a model using file parameters the user can use the command open [config] [positions] [robot] to render it to screen (note that order does matter here.</p> <p>Once a model is rendered they can once again use the console by pressing the i key they can then use the focus, unfocus, hide, show, clear and rebuild commands to drill down on specific robots within the environment and possibly remake them.</p> <p>The user can also save a currently rendered model by using the text entry box at the top left of the GUI by typing a name and pressing [Enter].</p> <p>The user can also navigate through the environment and zoom in & out using the key bindings listed in the user manual.</p>	
Alternative 1	If a user gives incorrect file paths, a file does not exist or the files given are not consistent or in the wrong order, the GUI will load with an error message telling the user what went wrong. They can then either close the application or retry running the model using the console.	
Alternative 2	If a user has press i they are in console mode and cannot use the key bindings to navigate through the current environment.	
Alternative 3	If a user tries to save a model with a name that already exists, the older version is overwritten without a warning.	
Alternative 4	If a user tries to load or open a model while a model is already on screen, the current model will be cleared and the new model displayed.	

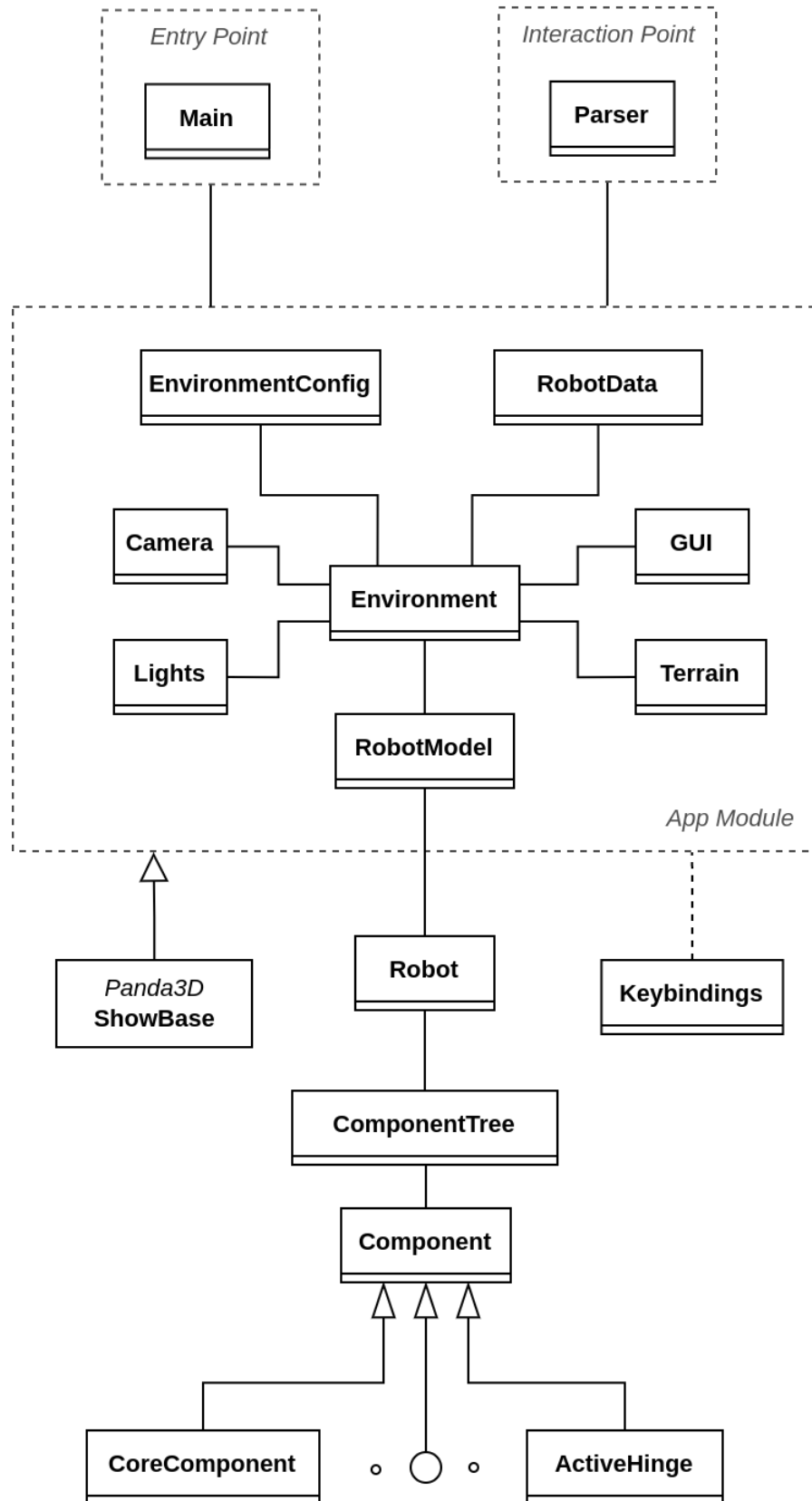


Fig. 2 (Simplified) Class Diagram

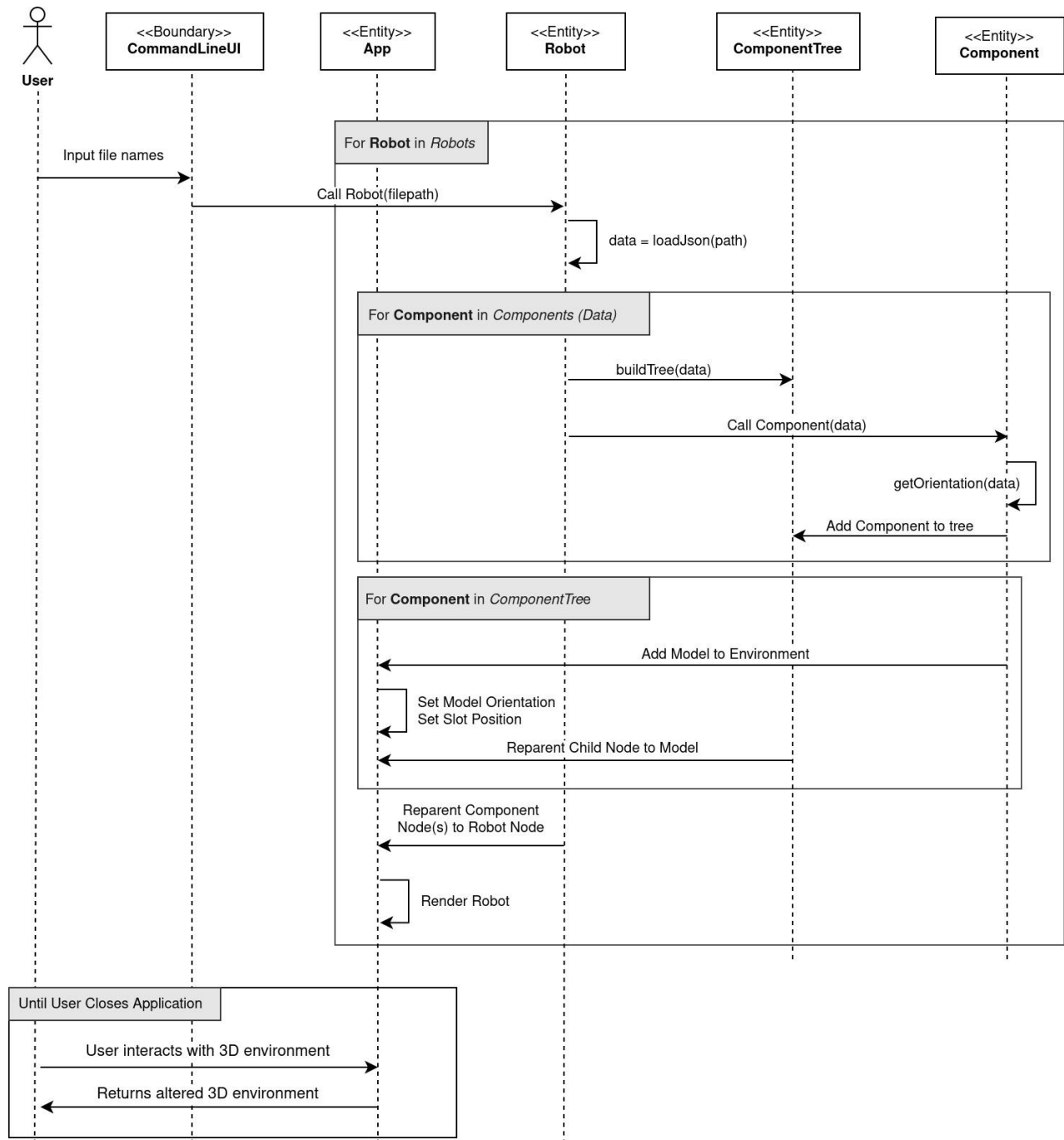

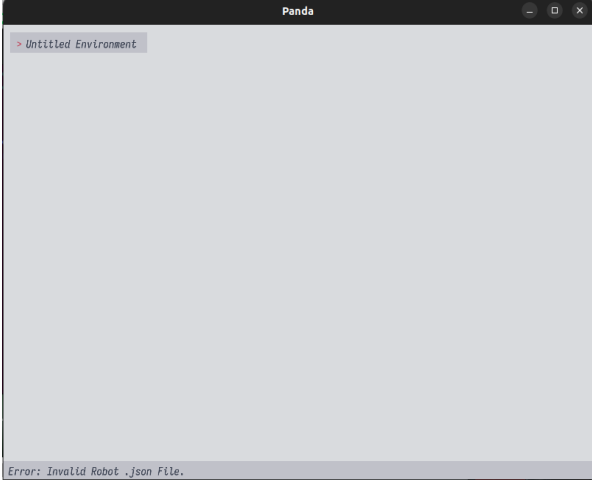

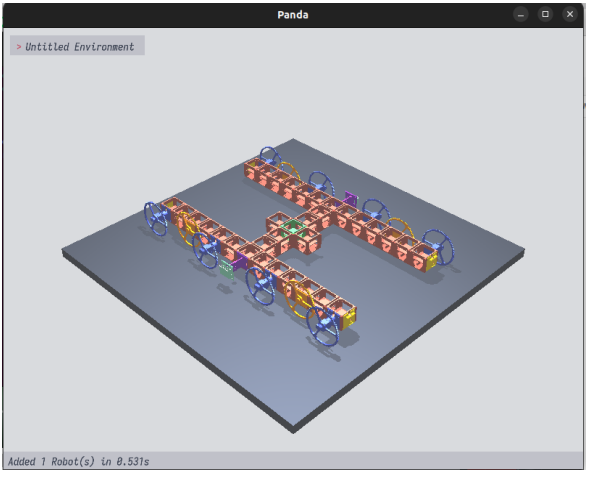


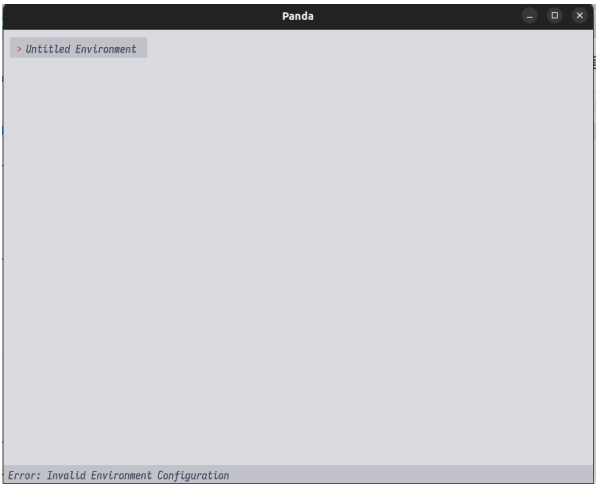
Fig. 4 Sequence diagram showing the model rendering process

Validation Test Examples

Below are a few examples of validation tests carried out and how the system dealt with them in the form of a screenshot during operation.

Case	User input	System output
Incorrect robot json file entered by user		
Correct file paths entered for config, position files and robot json files		

Incorrect file paths
entered for config and
position files



Missing file(file path)
from user input



User Manual (See README.md for markdown version)

Getting Started

Welcome to the RoboViz application. The purpose of this project is to provide robotics researchers with a local 3D visualisation tool which will allow them to simulate robotic swarms without large investment into physical experiments. Our intention is to provide an approachable, lightweight and customisable tool which does not seek to replace the use of RoboGen but instead provide a stepping stone into the world of robotics visualisation.

Modes of Use

As discussed earlier, our intention was to design the RoboViz application to be used by both experienced robotics researchers as well as those new to the field, possibly even computing in some respects. Because of this we have created 2 modes of use. Firstly a command line interface, that allows quick access to the applications features through various command line flags and parameters. Secondly, a basic GUI for users who might be uncomfortable using the command line to interact with the application.

Installing and Initialising RoboViz

After downloading the RoboViz repo, run the command **make install** to initialise the python virtual environment. This will ensure all dependencies and necessary packages are installed within the current folder location. Once installed, run the command **. venv/bin/activate** to activate the virtual environment.

Command Line Interface

Once the virtual environment is active you will have access to all available RoboViz functionality. Run the command **roboviz -h** to have available command line arguments printed to the terminal. These commands can be used to run specific features within the RoboViz application without having to bring up the GUI. Below we list a few interesting combinations to copy-paste for your convenience.

Command	Description
roboviz -c config/single/config.txt -p config/single/positions.txt -d data/cart.json	This will load a single instance of the cart type model. Note that order of each file does not matter, but must be preceded by the appropriate flag (-c, -p or -d).
roboviz -l	This will print available saved models to the terminal.
roboviz -L starfish	This will load a saved model, in this case the starfish.

roboviz -C help	The -C flag enables you to run GUI commands from the terminal. In this case, the GUI will be rendered and available commands printed to the console.
roboviz -c config/stress/single/config900.txt -p config/stress/single/positions900.txt -d data/cart.json	This will replicate one of our stress tests with 900 robots. Other available numbers to test your machine are 10000 and 90000. To run these make sure to edit the necessary numbers in the config and positions file paths.

Graphical User Interface

Using the console

If you wish to use the GUI instead of relying on the command line parameters, you can just run the command **roboviz** if you have correctly activated the virtual environment (see section on getting started). This will bring up the RoboViz GUI with an empty *environment*. Pressing **i** will activate the console across the bottom of the application (Note this must be pressed each time you want to enter a command into the console). This is the primary access a user has to RoboViz functionality when using the GUI. Typing **help** and pressing **enter** will list the available console commands. Below we list a few(with their descriptions and how to use them) to try out.

Console Command	Description	Usage
clear	Removes all robots and terrain and closes the environment	<i>clear</i>
save	Saves the current environment to disk as a Pickle file	<i>save [name]</i>
load	Loads an environment by name (searched in data directory)	<i>load [name]</i>
list	Lists all the saved environments	<i>list</i>
open	Open a model given the configuration file parameters	<i>open [config-path] [position-path] [data-path]</i>
focus	Focus on a robot given an id.	<i>focus [robot-id]</i>
unfocus	Display all robots	<i>unfocus</i>
hide	Hides all rendered objects.	<i>hide</i>

show	Show all rendered objects	<i>show</i>
rebuild	Rebuilds the environment under the current configuration	<i>rebuild</i>

Saving and Loading

Once the RoboViz GUI has loaded you will notice “Untitled Environment” at the top left. This is where models can be **saved** by entering a new name and pressing **enter** - the console will respond noting the model has been saved. You can view previously saved models with the command **list**. If you would like to pull up one of these models use the command **load [model name]**.

Interacting with a Rendered Model

Once a model is rendered you can interact with it by using the following key bindings.

KEY	USE	KEY	USE
-	Zoom out	i	Activate console input box in the app
=	Zoom in	Control-l	Clear the UI console
k	Move camera upwards	Shift-enter	Focus onto UI console
j	Move camera downwards	q	Exit app
h	Rotate camera to left	x	Clear environment screen
l	Rotate camera to right	Escape	Exit UI console
Arrow up	Move camera upwards	Tab	Autocomplete user entry
Arrow down	Move camera downwards	w	Move in positive y- axis direction
Arrow left	Rotate camera to left	a	Move in negative x-axis direction
Arrow right	Rotate camera to left	s	Move in negative y-axis direction
r	Reset camera to default position	d	Move in positive x- axis direction
Shift-r	Rebuild current system	Control -j	Zoom out

Responsibility Breakdown

Below is a comprehensive breakdown of responsibility within the project.

Section	Member
File Parsing	Jonty Doyle
Model Pipeline	Jonty Doyle & Hamza Amir
Command Line Interface	Ben Chiddy & Jonty Doyle
Graphical User Interface	Ben Chiddy & Jonty Doyle
Environment & Terrain Building	Ben Chiddy & Jonty Doyle
Util Class for Colour Handling	Hamza Amir & Jonty Doyle
Robot Building & Placing	Hamza Amir & Jonty Doyle
New Config & Positions Scripting	Hamza Amir & Ben Chiddy
New Robot Building	Hamza Amir
Stress Testing	Ben Chiddy
Validation Testing	Ben Chiddy, Hamza Amir & Jonty Doyle
Camera Controls & Key Bindings	Hamza Amir
Console Commands	Jonty Doyle & Ben Chiddy
Model Saving & Loading	Jonty Doyle
User Manual	Ben Chiddy & Hamza Amir
Final Report	Ben Chiddy, Hamza Amir & Jonty Doyle

References

Heiko Hamann (2019). *SWARM ROBOTICS: a formal approach*.

Laboratory of Intelligent Systems at EPFL (n.d.). <https://robogen.org/>. [online]

Available at: <https://robogen.org/> [Accessed 26 Sep. 2022].

www.panda3d.org. (2018). Panda3D | Open Source Framework for 3D Rendering & Games. [online]

Available at: <https://www.panda3d.org/>. [Accessed 26 Sep. 2022]