

University of Sheffield

# Increasing the usability of FLAMEGPU2 through the implementation of a GUI



Alex Parr

*Supervisor:* Robert Chisholm

A report submitted in fulfilment of the requirements  
for the degree of BSc in Computer Science

*in the*

Department of Computer Science

July 13, 2022

## Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: Alex Parr

---

Signature: A.Parr

---

Date: 10/05/2022

---

## Acknowledgements

I would like to thank my supervisor Robert Chisholm for his support and help in all aspects of this dissertation.

I would also like to thank my family and friends for their continued support and encouragement throughout my whole degree.

## **Abstract**

Agent-based modelling is an extremely powerful and useful tool that people in research and industry can use to more fully explain the behaviour of systems. FLAMEGPU2 is a framework that executes agent-based models directly on a computer's GPU, offering significant performance gains over similar frameworks and applications.

Traditionally, a relatively high level of programming knowledge is needed to be able to implement such models in research projects. This report details how the barrier of entry to the FLAMEGPU2 framework has been reduced by providing users with an intuitive GUI which allows them to define both the model structure and behaviour visually rather than through code. This will allow domain experts to gain the benefits of creating high performance agent-based models, without requiring advanced programming skills.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Overview of the Report . . . . .	2
1.2	Relationship to Degree Program . . . . .	2
<b>2</b>	<b>Literature Review</b>	<b>3</b>
2.1	What is an Agent-Based Model . . . . .	3
2.2	Uses for Agent-Based Modelling . . . . .	3
2.3	FLAMEGPU2 . . . . .	4
2.3.1	What is FLAMEGPU2 . . . . .	4
2.3.2	How is FLAMEGPU2's Approach Unique . . . . .	4
2.4	Pre-existing GUI Implementations of Agent-Based Simulations . . . . .	5
2.5	Human Computer Interaction . . . . .	8
2.6	Modules/Libraries . . . . .	9
2.6.1	Creating the GUI . . . . .	9
2.6.2	Code Generation . . . . .	12
<b>3</b>	<b>Requirements and Analysis</b>	<b>14</b>
3.1	Aims and Objectives . . . . .	14
3.2	Design Inspiration . . . . .	14
3.3	Technology Selection . . . . .	15
3.4	Project Requirements . . . . .	16
3.5	Evaluation . . . . .	16
<b>4</b>	<b>Design</b>	<b>18</b>
4.1	Overview . . . . .	18
4.2	Architecture . . . . .	18
4.3	Mock-Up Designs . . . . .	20
4.3.1	Main Window . . . . .	20
4.3.2	Message Dialog . . . . .	22
4.3.3	Agent Creation Dialog . . . . .	23
4.3.4	Configuration Dialog . . . . .	24

<b>5</b>	<b>Implementation and Testing</b>	<b>25</b>
5.1	Introduction . . . . .	25
5.2	Application Creation . . . . .	25
5.2.1	Creating the UI . . . . .	25
5.2.2	Interface Behaviour . . . . .	26
5.2.3	Validation . . . . .	28
5.2.4	Code Generation . . . . .	29
5.3	Changes Made During Development . . . . .	30
5.3.1	Visual changes . . . . .	30
5.3.2	Additional Features . . . . .	30
5.4	Testing . . . . .	31
5.4.1	Integration Testing . . . . .	31
5.4.2	End-To-End Testing . . . . .	33
<b>6</b>	<b>Results and Discussion</b>	<b>37</b>
6.1	Outcome . . . . .	37
6.2	Evaluation of Application . . . . .	37
6.2.1	Meeting of Requirements . . . . .	37
6.2.2	Evaluation . . . . .	39
6.3	Further Work . . . . .	39
6.3.1	Extensions to This Project . . . . .	39
6.3.2	Ideas For New Projects . . . . .	40
<b>7</b>	<b>Conclusion</b>	<b>42</b>
	<b>Appendices</b>	<b>46</b>
<b>A</b>	<b>Tkinter Sample Code</b>	<b>47</b>
<b>B</b>	<b>PyQt6 Sample Code</b>	<b>48</b>
<b>C</b>	<b>Kivy Sample Code</b>	<b>49</b>
<b>D</b>	<b>WxPython Sample Code</b>	<b>51</b>
<b>E</b>	<b>Application Screenshots</b>	<b>53</b>
<b>F</b>	<b>Testing</b>	<b>57</b>
<b>G</b>	<b>Test Code</b>	<b>59</b>

# List of Figures

2.1	An example of the NetLogo GUI [11] . . . . .	6
2.2	SOARS VisualShell uses a drag and drop feature to create new agents, roles and spots[7] . . . . .	7
2.3	Repast Symphony[12] lets users draw statecharts for their agents . . . . .	8
2.4	An example window using Tkinter (See Appendix A for source code) . . . . .	10
2.5	An example window using PyQt6 (See Appendix B for source code) . . . . .	10
2.6	An example window using Kivy (See Appendix C for source code) . . . . .	11
2.7	An example window using WxPython (See Appendix D for source code) . . . . .	12
4.1	Flowchart showing the progression through the system . . . . .	19
4.2	Proposed design for Main Screen . . . . .	21
4.3	Proposed design for Message Dialog . . . . .	22
4.4	Proposed design for Agent Creation Dialog . . . . .	23
4.5	Proposed design for Configuration Dialog . . . . .	24
5.1	An annotated screenshot of the final application . . . . .	26
5.2	The link the user can make between agents and their functions . . . . .	27
5.3	Message Dialog showing default values for a SpatialMessage3D . . . . .	31
5.4	Simulation at the initial stage . . . . .	34
5.5	Simulation after agents start to congregate . . . . .	34
5.6	Simulation after agents form first spheres . . . . .	35
5.7	Final stage of simulation where agents find equilibrium . . . . .	35
E.1	Screenshot of the final application with a basic model created using it . . . . .	53
E.2	Final Agent Dialog showing 3 sample variables added . . . . .	54
E.3	Final Message Dialog showing one message already created and 2 sample variables added . . . . .	55
E.4	Final Configuration Dialog to setup how the simulation will run . . . . .	55
E.5	The control flow panel when multiple agents are linked to the same function . . . . .	56

# List of Tables

3.1	Comparison of Python GUI modules . . . . .	15
3.2	List of proposed application requirements . . . . .	16
5.1	Integration testing for Main Window . . . . .	32
5.2	Integration testing for Agent Dialog . . . . .	32
5.3	Integration testing for Message Dialog . . . . .	33
5.4	Integration testing for Configuration Dialog . . . . .	33
6.1	Requirements Achieved . . . . .	38
F.1	End to End testing of application . . . . .	57
F.2	End to End testing of application (continued) . . . . .	58



# List of Algorithms

1	A class able to generate Python scripts and save them . . . . .	13
2	The class used to generate the Python FLAMEGPU2 script . . . . .	29
3	“output_message” agent function code . . . . .	59
4	“move” agent function code . . . . .	60
5	“Validation” host function code . . . . .	61

# Chapter 1

## Introduction

Simulation is a powerful and valuable technique that can improve understanding in a given situation. It is used in a wide range of applications from modelling large crowd movements to the growth of a cancerous tumour. Simulation of a real world problem is useful as it can explain the reasons for things occurring, and as a result be able to predict what the outcome of a given situation might be without having to carry out the test in the physical world. This is beneficial in areas where a physical experiment may be impractical due to, for example, cost or rarity of occurrence. A simulation provides a safe and inexpensive environment to test and observe theories, and in a repeatable way. FLAMEGPU2 is a particularly powerful agent-based modelling (ABM) framework which runs the simulation on the computer's GPU instead of CPU, offering large performance advantages due to the GPU running in a more highly parallel way than a CPU.

To create an ABM simulation with FLAMEGPU2 containing hundreds of thousands of agents in a 3D space, a strong knowledge of either C++ or Python is required as well as proficiency with how FLAMEGPU2 scripts are structured. This means even when these kinds of simulations are implemented in research, two situations can occur; either a programmer without any specialised knowledge of the area of research writes the simulation, or the expert conducting the research writes it themselves. If a dedicated programmer is employed to create the simulation then the behaviour and characteristics of the simulation must be conveyed to them which may be difficult or take a large amount of time. However if the researchers themselves create it, then they have to take time to learn the necessary skills taking up more time and pulling their focus from the research itself.

This project aims to offer a way for people with very limited or no programming experience to be able to define ABM simulations using FLAMEGPU2 without explicit knowledge of how FLAMEGPU2 scripts are structured or how to program. Multiple executions of graphical editors for ABMs have been investigated and informed decisions in the process of developing a GUI based solution. The solution has been shown to reduce the amount of knowledge required for defining a model and therefore will open up the benefits of using high performance ABMs to many more people and allow it to be used in a wider range of fields.

## 1.1 Overview of the Report

In this report Chapter 2 investigates the current technologies in relation to this project including; pre-existing ABM software, human computer interaction (HCI) and Python modules that may be relevant to the development of this project. Chapter 3 then details the aims of the project, listing distinct project requirements and how they are measured and tested for. The next chapter gives the design for the proposed system including mock-ups of each screen. How the program is actually be made is described in Chapter 5 as well as the testing carried out. The evaluation of the system along with what has been learnt throughout the project is given in Chapter 6. In addition any new areas of research which could come from this project are given in this chapter. The final chapter gives a summary of the project as a whole and what has been learnt from it.

## 1.2 Relationship to Degree Program

The technical knowledge required for undertaking this project is mostly split between two areas; agent-based modelling and building GUIs. The Computer Science Undergraduate Degree offered at The University of Sheffield covers both of these in part. Two modules undertaken by the author have provided teaching on ABMs including one assessed assignment on it. And while multiple projects over the degree have included the need for creating a GUI, the principles behind them have never been explicitly taught. This overall has provided a baseline knowledge of these areas while still allowing for abundant expansion of understanding.

## Chapter 2

# Literature Review

### 2.1 What is an Agent-Based Model

Agent based modelling is the method of creating a simulation containing (usually) a large number of copies of “agents”. These agents execute functions that describe their behaviour, often in relation to their movement or internal property states. Within an agent-based model, there may be a few different types of agents to represent different entities in the simulation but for it to be a true ABM, the emphasis must be on there being multiple copies of the same type of agent that therefore execute the same functions. The reason why this is such a powerful approach is that these types of simulations can result in showing emergent behaviour which would not have been predictable with other techniques such as equation-based modelling.

### 2.2 Uses for Agent-Based Modelling

Agent-based modelling is used in many different areas of research and industry. As the models consist of a large number of autonomous entities, complex emergent behaviour can arise and be learnt from through observing the interactions between agents. This behaviour of the system as a whole is often unexpected and counter intuitive, showing how valuable ABMs can be. Modelling the system as one unit would display incorrect behaviour and therefore not be suitable as it would include inaccurate assumptions. Tests or experiments which are impractical in the real world are much more convenient to implement in an ABM as the cost and difficulty is often largely reduced.

In a paper by McLane et al[10], ecologists implemented an agent-based model to come to an informed decision about which habitats to protect and designate for use by wildlife species. The conclusion they came to then advised policy makers in which areas to protect, while taking into account potential economic loss by not using the land. The use of an ABM in this situation was particularly helpful as it provided a large amount of flexibility in what they were able to simulate and the high level of detail they were able to attain.

ABMs have also been used to implement crowd simulations[2] but with the distinct difference that the information within the simulation was being updated in real-time. This

allows the model to be used in managing crowds in real-time rather than running simulations beforehand from a predictive viewpoint.

## 2.3 FLAMEGPU2

### 2.3.1 What is FLAMEGPU2

FLAMEGPU2 (FLexible Agent Modelling Environment for GPUs) is a framework that allows users to write easily understandable high-level C++ or Python to create and run ABMs which execute directly on the computer’s GPU. FLAMEGPU2 was created by a research group at the University of Sheffield in 2021 lead by Paul Richmond[13]. This second iteration of the software was a complete re-write, improving usability and allowing scripting in Python as well as C++, unlike the first iteration which only offered C++ support. This has provided much more flexibility for users and opened up the use of it to many more people. By using this framework, people can create complex and powerful simulations that run in real-time without specific knowledge of writing code optimised for a GPU.

The framework and the models created within it consist of; agents, functions, environment properties and messages. Messages are formats for how information is transferred from function to function and agent to agent. When defined, they specify the variables each message of that type will have as well as the method of which agents can access them (e.g based on where they are spatially). Environment properties act as global variables which can be accessed anywhere throughout the model. Agents are then defined with properties and linked to functions that describe their behaviour in the simulation. These functions take as input and output one of the pre-defined messages. Multiple different types of agents can be specified, each with unique behaviours which can all interact with one another. The agent functions execute in an order specified in one of two ways, either by organising them into layers or adding them to a dependency graph. If using the layers approach, it is explicitly stated which function executes after which other. However using the dependency graph (which was added in the second iteration of FLAMEGPU) offers a higher level of abstraction where the user specifies which function depends on which, and from that, FLAMEGPU itself constructs the exact order. The simulation can then be run and values such as agent populations or environment properties can be extracted at each step to then be displayed and analysed afterwards.

### 2.3.2 How is FLAMEGPU2’s Approach Unique

FLAMEGPU2’s advantages come from it executing the code for agent behaviour directly on the computer’s GPU which offers a huge benefit due to a GPU’s architecture in comparison to a CPU (which most agent-based simulations are run on). GPUs contain orders of magnitude more computing cores than a CPU and those cores execute in a highly parallel fashion. FLAMEGPU2 requires an Nvidia GPU to run, utilising CUDA (Compute Unified Device Architecture) to interface with the GPU directly. Running ABMs on the GPU makes use

of the SIMT (Single Instruction Multiple Threads) architecture as the same code is run for every agent of the same type, allowing them all to be computed simultaneously, reducing the time it takes to compute and increasing performance.[1].

The advantages of this is that simulations using FLAMEGPU2 can contain massive populations of agents, all simulated in real-time as the performance benefits are so dramatic. Simulations of agent numbers in the order of hundreds of thousands or more are possible, allowing for more complex simulations to be run. In addition, the framework also allows for a real-time visualisation of the simulation to be produced, as the agent data is already located on the GPU.

## 2.4 Pre-existing GUI Implementations of Agent-Based Simulations

There are already examples of some systems that make use of a GUI to define agent-based models. While each take a slightly different approach, things can be learnt from all of them.

NetLogo[11] offers users a GUI based solution to creating agent-based models. While it does still require the users to define most of the characteristics of the simulation in code, it does abstract out some control over the simulation behaviour to the GUI (shown in Figure 2.1). It offers a more interactive version of simulation in comparison to FLAMEGPU2 as the GUI elements let the user alter the simulation such as changing variable values as the simulation executes. The GUI also allows users to select certain variables to view their values in a graph over the period of the simulation. Like FLAMEGPU2, NetLogo can visualise ABM simulations up to 3 dimensions however it differs from FLAMEGPU2 as higher dimensional models cannot be implemented. NetLogo is very effective at creating simple models of situations and observing the impact of different parameters on it, however due to it being a single threaded CPU application, it does not scale well when large agent populations are involved.

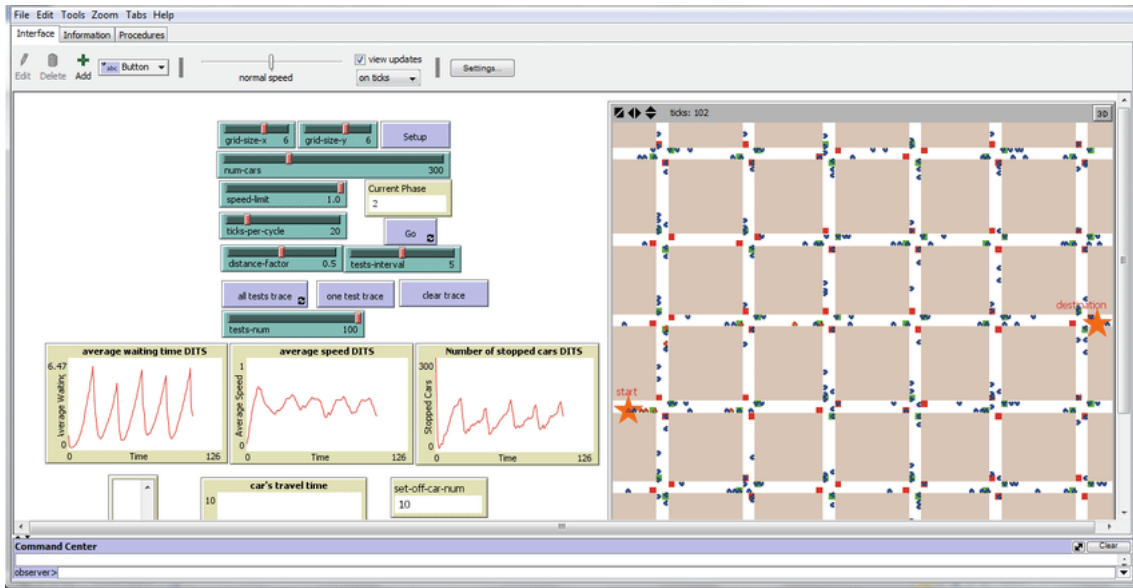


Figure 2.1: An example of the NetLogo GUI [11]

In a 2015 paper[7], a group developed another GUI to create ABMs. It was for the software SOARS (Spot Oriented Agent Role Simulator), and it aimed to increase the reach of the software by providing a way to create simulations using it without the need to be able to program. In SOARS, agents containing variables can move between 'spots' that can also contain variables. The movement of the agents is defined by a series of agent and spot roles containing conditions for certain movements. In the paper they were able to construct a GUI that fully encompassed all of SOARS' features via dragging and dropping elements to create a diagram (shown in Figure 2.2) that describes the model. This is a more full abstraction as the proposed solution in this paper will still require some code to be written by the user. However, the lack of the ability to write any code does mean the range of logic that can be defined and executed in each role is limited. This is overall a much weaker solution than both FLAMEGPU2 and NetLogo with applications being much more limited as the model does not exist in any dimension. Another way it differs from the other two is it does not provide any visualisation of the model as it executes.

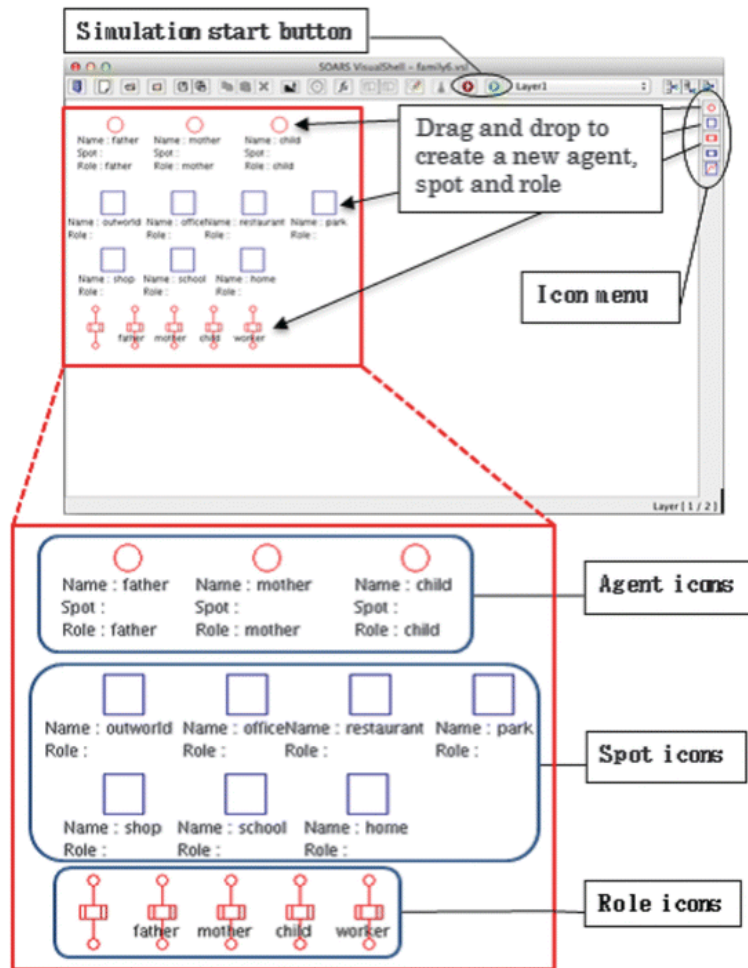


Figure 2.2: SOARS VisualShell uses a drag and drop feature to create new agents, roles and spots[7]

A widely used agent-based modelling system, Repast, has a GUI tool (Figure 2.3) to help in creating the models called Repast Symphony[12]. It gives users options as to how they wish to define the model with a combination of ReLogo, Groovy, Java or visually through statecharts. While the use of the statecharts do mean the user can specify behaviour without coding, some coding is still needed to link the them to the model. Repast provides a visualisation for the simulations it creates, just like FLAMEGPU2 and NetLogo. While Repast is multi-threaded, as it does not execute on the GPU it is at a distinct disadvantage to FLAMEGPU2 when situations where large numbers of agents are required.



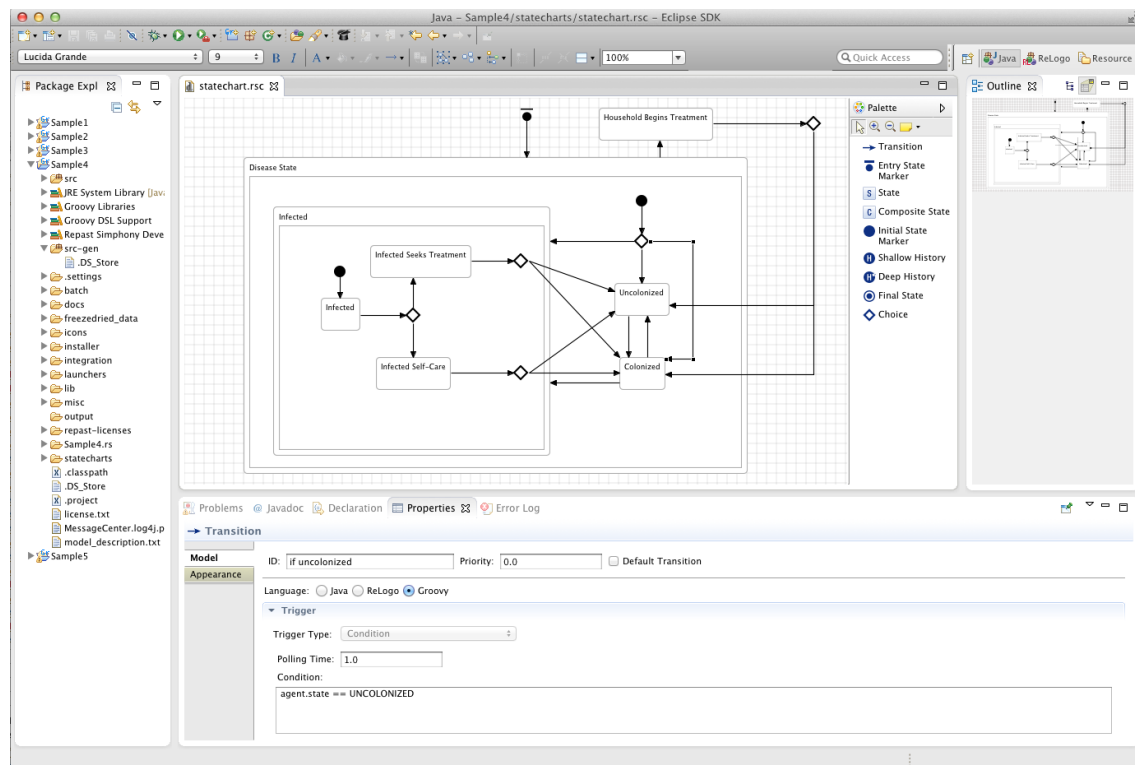


Figure 2.3: Repast Symphony[12] lets users draw statecharts for their agents

## 2.5 Human Computer Interaction

How users interact with a computer system is a key part of how positive their experience with it is. The study of how applications convey information to the user is known as Human Computer Interaction (HCI) and should be an important part in any UI design project. By using techniques that make interaction with an application as seamless as possible, users can get the best outcome from their experience with it.

A study by Dyck et al[4] into the techniques that video games use in creating their interfaces and how they can be applied to other kinds of applications, found the prominence of “Calm Messages” in video games. These are messages that appear to the user but do not require direct interaction with to dismiss, therefore keeping the user engaged in the main use of the system. Messages can take this format if they are not at a high level of priority, and purely for the purpose of giving the user some extra information. Major messages that inform the user on things that might interrupt their current task in the application still use a traditional dialog box which needs to be dismissed. The use of this technique in applications outside of video games could provide an overall better user experience for people interacting with the system.

Wilbert O Galitz[5] investigates how user interfaces are designed and evaluates the features

that make them up in a book published in 2007. It explains the benefits of an interface design known as “Direct Manipulation”. A direct manipulation interface is when the user can interact directly with the elements on the screen and many benefits are given in the book including it leading to faster learning and making the interface easier to remember. Additional styles of interface are also given including “Form Fill-In” which provides an easy way for information to be input into the system.

## 2.6 Modules/Libraries

The GUI application for this project will be written in Python due to the author already having experience in the language and the vast number of modules it has which massively extend the features of the language. Making good use of these can reduce development time for projects. This section investigates the possible modules that could be relevant to this project and some of their advantages and disadvantages.

### 2.6.1 Creating the GUI

A strong GUI building module allows the developer to define windows and add widgets (such as buttons, labels, etc) to create the user interface in an easy way. Some are made for creating specific types of interfaces and others are used in a more general way. The modules Tkinter[9], PyQt6[8], Kivy[15] and WxPython[3] will be considered. For all modules a sample window has been created each with an example of a label, button and text entry box.

Tkinter is the standard GUI building library for Python; with it already being built into Python itself, it doesn’t require any additional installation. It runs across Linux, Windows and macOS, however the windows it creates and the elements within the window are not styled by default in a way which makes it look native to the operating system. Tkinter makes it easy to create basic windows showing simple content however from previous experience, it is found to be cumbersome to format complex windows with lots of elements in. Tkinter resides under the same licence as Python itself so it can be freely used by anyone.

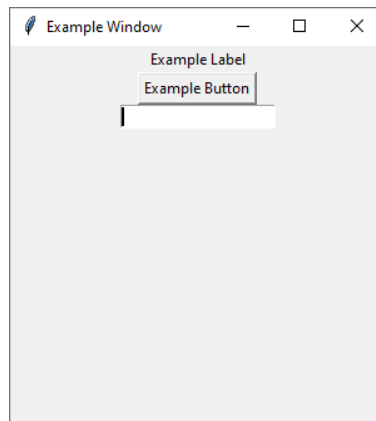


Figure 2.4: An example window using Tkinter (See Appendix A for source code)

A powerful alternative GUI building module is PyQt6. The library is a binding for the Qt toolkit, enabling GUIs to be created for a wide range of operating systems, much like Tkinter. PyQt6 allows users to write the UIs in Python themselves but also offers a drag and drop interface where developers can visually move elements around, making it easier to create more complex windows, something that other GUI building modules do not offer. The elements that the module renders are styled to the operating system it is running on, giving a professional look to the window that Tkinter does not. PyQt6 is under two licences, GPL v3 and Riverbank Commercial License. As long as commercial gain is not being taken, the open and free GPL v3 licence is sufficient. The size of the module is 6.3MB which is not large enough to cause an issue however it is not as convenient as Tkinter being pre-installed.

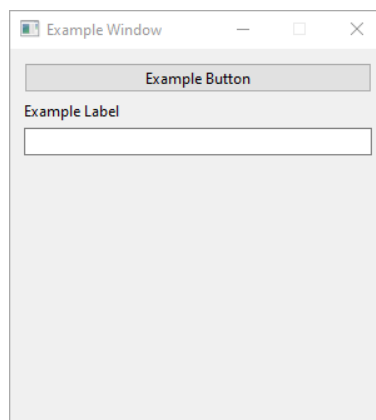


Figure 2.5: An example window using PyQt6 (See Appendix B for source code)

Kivy is a Python module which is often used to create Android and iOS applications as it supports touch inputs but can be used for PC based apps in addition. Like the other modules investigated it supports all of the 3 main desktop operating systems, however it does

not apply any styling to the on-screen elements by default as it is aimed more towards game development where heavy styling will be applied by the developer. As a result the interfaces Kivy produces do not have a professional feeling as they would if they were natively styled. This puts it at a distinct disadvantage to Tkinter and PyQt6. Kivy also requires additional .kv files to specify the contents of windows and their styling, adding additional complexity over the other module choices. It is distributed under the MIT License meaning it is open source and can be used freely. Its download size is marginally better than PyQt6 at 4.1MB but this difference is negligible.

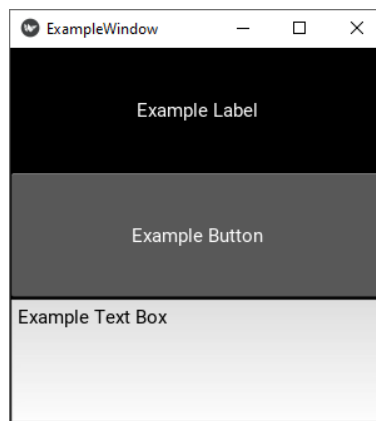


Figure 2.6: An example window using Kivy (See Appendix C for source code)

Finally, WxPython is another GUI building module which offers a wide selection of widgets available for developers to pick from. However this does come with the drawback of being the largest of the modules researched at 18.1MB. WxPython is available on all main operating systems and the windows that the module produces have a default native styling applied depending on the operating system, just as PyQt6 does. The module is with a OSI Approved wxWindows Library Licence meaning it is open source as the other 3 modules are too.

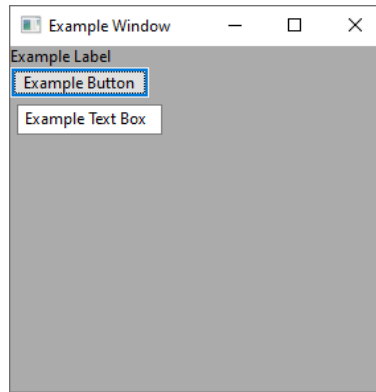


Figure 2.7: An example window using WxPython (See Appendix D for source code)

### 2.6.2 Code Generation

Within the system proposed, there must be a way of generating the Python code that will use the FLAMEGPU2 framework to run the simulation from the GUI. This is so that the resulting code can be run without the use of the GUI and could be viewed completely separately to make changes. The solution chosen must be efficient, creating the script ideally in just a few seconds so that it does not act as a bottleneck in the system.

Jinja2[14] is a module that developers can use to generate code based on a predefined template. It is usually used for creating HTML files but can be used to populate any type of file. This means large amounts of boilerplate code can be written in advance and saved to a text file with markers to be replaced with specific data. The download for the module is very small (about 133kB) so doesn't cause any inconvenience. It is under a BSD Licence which therefore does not have any requirements for redistribution.

Another option is to not use an existing module from the Python library but to write a bespoke class that will handle generating a Python script. This is advantageous over using Jinja2 as there would be fewer dependencies to end users, reducing the number of points where errors may occur during installation and reducing overall package size. The features implemented to the class could be kept to the very minimal of what is needed, creating a more lightweight and flexible solution from which features could be added as needed. This approach would also mean no time would have to be taken to learn a new module and how it works before development of that section, however time would have to be spent writing the bespoke class itself. It is unclear if this would take longer overall. As Jinja2 is a well established module with a large number of users, it would likely be a more robust solution, delivering a more efficient code generator and with a lower likelihood of containing bugs. Due to it being used by many people, the support community behind is also substantial, offering potential help which would not be available with the bespoke solution.

---

**Algorithm 1** A class able to generate Python scripts and save them

---

```
class codeGen:

    def __init__(self, tab="\t"):
        self.code = []
        self.tab = tab
        self.level = 0

    def write(self, lines, indent = 0):
        for line in lines:
            self.code.append(self.tab * self.level + line)
        self.indent(indent)

    def indent(self, indent = 1):
        if self.level + indent >= 0:
            self.level = self.level + indent
        else:
            print("Indent Error")

    def save(self, filename):
        f = open(filename, 'w')
        f.write("\n".join(self.code))
        f.close()
```

---

The sample code in Algorithm 1 shows that only minimal amounts of code is required to create a class that could generate Python source code and save it to a file.

## Chapter 3

# Requirements and Analysis

### 3.1 Aims and Objectives

This project aims to reduce the level of technical knowledge required and put the process of creating ABM simulations into a more understandable form to open up the benefits to a much wider number of people. This will be done by creating a GUI for the agent based modelling framework FLAMEGPU2 which will provide a more visual way of defining the model and functionality of the simulation, simplifying the development process. Some coding will still be required for the user to specify the agent function logic as finding a way of having this in graphical form is infeasible and out of this project scope.

The final user interface is intended to allow users to specify all aspects of the simulation including agents, messages, agent functions and environment properties. When the user then indicates to run the simulation, the program will then generate a Python script that implements FLAMEGPU2 to execute the simulation and run it.

If this project is successful in lowering the barrier of entry for using FLAMEGPU2 then it could be distributed as an addition to FLAMEGPU2 to then be used by anyone.

### 3.2 Design Inspiration

From the research into the literature surrounding this project, some features stand out which would benefit the application from being included. The inclusion of these look to make the user's experience as positive as possible through good design.

Using "Calm Messages"[4] to tell the user information would be an effective way to communicate to the user parts of the system that are still to be defined, such as if there are functions that are yet to be connected to an agent. In this instance, it is something that the user should be aware of but not important enough to warrant stopping the user from what they are doing.

Repast's use of statecharts which users can visually draw out lends itself very well to this proposed application. Although you do not explicitly define any statecharts in FLAMEGPU, a similar system could be implemented for linking agents to their functions where the user

can drag in new agents and functions and then draw their links. This would be a universally understandable way of creating the key points of the model and provide an easy way to communicate to others how the model is structured.

### 3.3 Technology Selection

From the literature review in Chapter 2, a decision has to be made as to which Python modules would be best suited for use in this project. Consideration about the requirements for this project need to be taken into account; with the hope this informed decision will provide the easiest tools during development.

The main areas to be considered when choosing the GUI building module are; the usability of the module so that it will not take too long to learn, the licensing it comes with so it is possible to be redistributed with FLAMEGPU2, and the features it provides so that development is not limited in this way.

Module	Download Size	Native Styling?	OSI Approved?
Tkinter	N/A	No	Yes
PyQt6	6.3MB	Yes	Yes
Kivy	4.1MB	No	Yes
WxPython	18.1MB	Yes	Yes

Table 3.1: Comparison of Python GUI modules

After exploring the different opportunities for GUI modules summarised in Table 3.1, PyQt6 seems to be the one most suited for this project for multiple reasons. Firstly its native styling will ensure the final product will look professional. Due to its powerful architecture and its wide range of elements it will be able to provide all the functionality required during development and PyQt6’s open source licence will also allow the project to be distributed should the GUI be used widely. While WxPython also has these features, PyQt6 is set apart as it provides a tool for building the GUI visually by dragging and dropping elements onto the window itself. This will save large amounts of time when building the system and is the reason why it will be used rather than WxPython.

For the selection of how code generation will be achieved, a bespoke code generator class would be best suited in this situation. The large amounts of flexibility it offers will be hugely beneficial as any additional features required can be implemented. In addition, in FLAMEGPU2 scripts, very little boilerplate code is needed and scripts can vary largely, therefore using a module such as Jinja2 which is very effective at swapping out tags in a larger script for specifics would not be as valuable. This choice will massively reduce the amount of time required to learn a new module, which will save time overall. This will deliver a very lightweight solution, with only the features that are required being included.



The reason a way to generate code and for it to be saved to an external file is required is that the application needs to be able to create a standalone complete FLAMEGPU2 script. If the model were to be defined within the application and executed from itself, then it would not be possible to run the simulation without the use of the interface.

### 3.4 Project Requirements

To guide the development of this project, a series of requirements given in Table 3.2 have been devised with help from the prior research. Each requirement is given priority of either mandatory (M), desirable (D) or optional (O). This gives a clear definition of the minimal viable product as well as the other features that would be very beneficial to have.

Number	Requirement	Priority
1	Define new ABM simulation elements including agents, messages, agent functions and environment properties through the GUI	M
2	Set simulation configuration such as number of steps it will run for and initial populations of agents	M
3	Specify execution order of agent functions by organising into layers	M
4	Run the simulation from the GUI	M
5	View a graph of agent populations over time after the simulation has executed	D
6	View a visualisation of the model as it executes	D
7	Generate and save a standalone Python source code file of the simulation	D
8	Define agent populations by attaching an XML file	D
9	Load pre-existing FLAMEGPU2 scripts into the GUI to view them	O
10	Change attributes of the visualisation (e.g agent models, colours, etc)	O
11	Define control flow through a dependency graph rather than layers	O
12	View any agent attribute over time in a graph	O

Table 3.2: List of proposed application requirements

### 3.5 Evaluation

Evaluation of the final system will go through two stages of testing. The first will be integration testing, ensuring that all the features that are required are present and work

as expected. The second will be an end-to-end testing where the application is used from start to finish as it would be expected to be used by the end user. This will indicate whether it can actually be used as intended.

The integration testing will methodically go through every feature of the application and test if it works as expected. This testing will make it much easier to access how many of the requirements of the project are met. These tests will not take place in a specific context, as it is just to see which component parts work.

The end-to-end testing will consist of implementing an ABM simulation using the application created, which will give a holistic view of the extent to which the aims of this project have been achieved. The simulation which will be implemented is the “Circle” example given in the FLAMEGPU2 GitHub repository. It consists of a large number of objects starting in random locations, and with each step they move towards each other to congregate into spheres of a given radius. This example had been chosen as it is complex enough to use all the parts of the system but not too complex such that it would take too long to implement.

# Chapter 4

## Design

### 4.1 Overview

The design of the UI for this project is arguably the most important section of this entire project as the aim is to increase the usability of FLAMEGPU2 and therefore the user experience needs to be as frictionless as possible. As this program is aimed to be used as a tool for researchers, the focus of designing the UI is to ensure that every feature is accessible in an easy and clear way as opposed to prioritising the overall aesthetics of the program. Therefore the layout will be practical and provide as much information to the user as possible.

### 4.2 Architecture

The application will follow a relatively simple sequence shown in Figure 4.1. It will start with the user using the UI to define the simulation including; agents and their attributes/functions, environment properties, messages and execution order. After this the application will extract all the defined data from the UI and use it to generate a Python script that will create the FLAMEGPU2 model which will be saved to a specified file location. This generated script will then immediately be executed with an optional visualisation of the simulation displayed if the user requested it. Once the simulation has finished, relevant graphs of population numbers/agent attributes will be displayed.

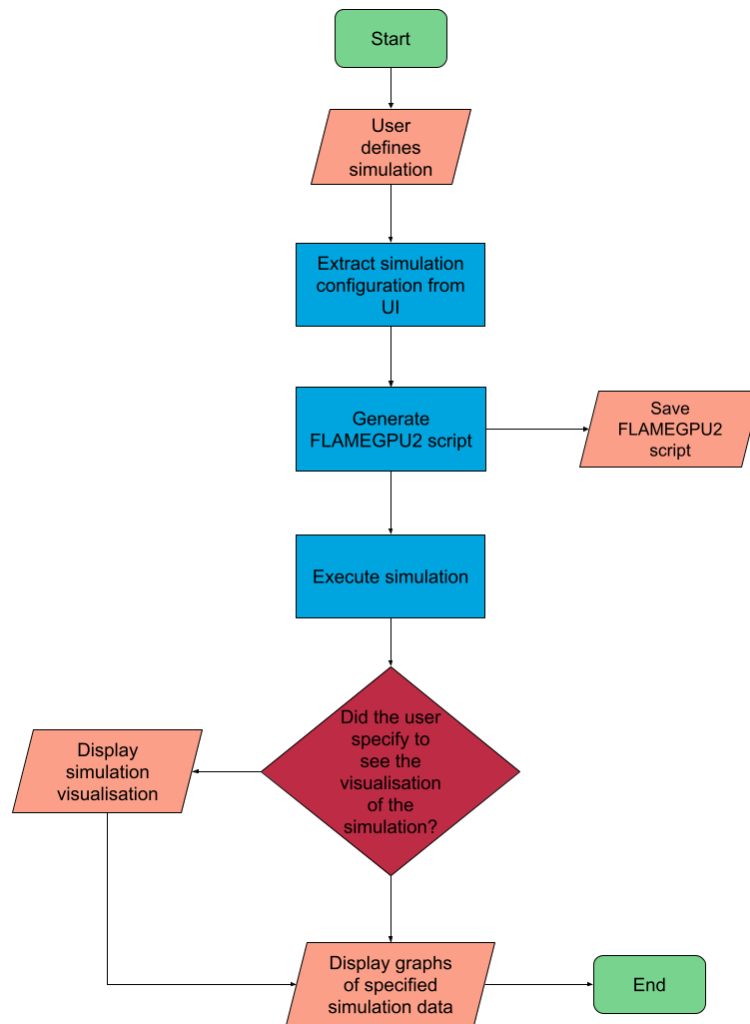


Figure 4.1: Flowchart showing the progression through the system

In order to structure the program to make adding features or someone understanding the code easier, it will use the Model-View-Controller (MVC) [6] design pattern. This works by separating different sections of the code based on what they do and while MVC is most often used when creating web application, its principles still work on standard desktop applications.

- **Model**

The MVC model files read and write data to the internal data structure. Usually this would be a database, however as this application does not require a database as it does not need to store large amounts of data in a centralised place, the model files interact with the data structures in play such as lists, dictionaries and JSON objects.

- **View**

The view files contain the code for defining how the user interface looks and acts, giving the location and properties of all the onscreen elements.

- **Controller**

The controllers then act as an intermediary for the models and views, transferring data from one to the other and completing any logic needed for the system such as validation.

## 4.3 Mock-Up Designs

Mock-ups of every screen have been created below to give a guide of what each should include when it comes to implementation. However as this is just a guide, some elements may change in the final version of the system if different ways of laying out elements is found to offer certain advantages.

### 4.3.1 Main Window

This Main Screen design as seen in Figure 4.2 meets requirements 1 and 3; giving the user access to environment properties, the order in which the simulation executes and the linking of agents and their behaviours. The main section in the middle will be scrollable both horizontally and vertically in case the simulation being created contains a large number of agents and/or agent functions so that the screen will not become too overcrowded and allows for scalability. In the top right corner (Figure 4.2 section B), users will be able to define the execution order of their simulation by dragging and dropping the function cards to reorder them, and adding new layers. The “Add Agent” button (Figure 4.2 button C) will be used to add new agents to the simulation (via a dialog window). The menu bar at the top (Figure 4.2 section A) giving access to menu items will allow the user to open, save, add new messages to the system and run the simulation once complete.

This screen is designed to provide an overview of the key parts of the simulation for the user. This is important as, should they want to show and explain the simulation to someone else, it will be easy to understand as it is contained in one screen and laid out in a logical way.

The agent and function blocks will show the key information to the user. The agent block will show the name of the agent and then list beneath the attribute names and types. It won't show the initial values of these attributes as it is non-essential for defining the rest of the model and would cause the agent blocks to become overcrowded and too large. The function blocks will similarly display their name at the top but then below will have two combo boxes giving the user the option to chose the message the function will use for its input and output. A large text box will then fill the rest of the block which will be used for the body of the function's code.

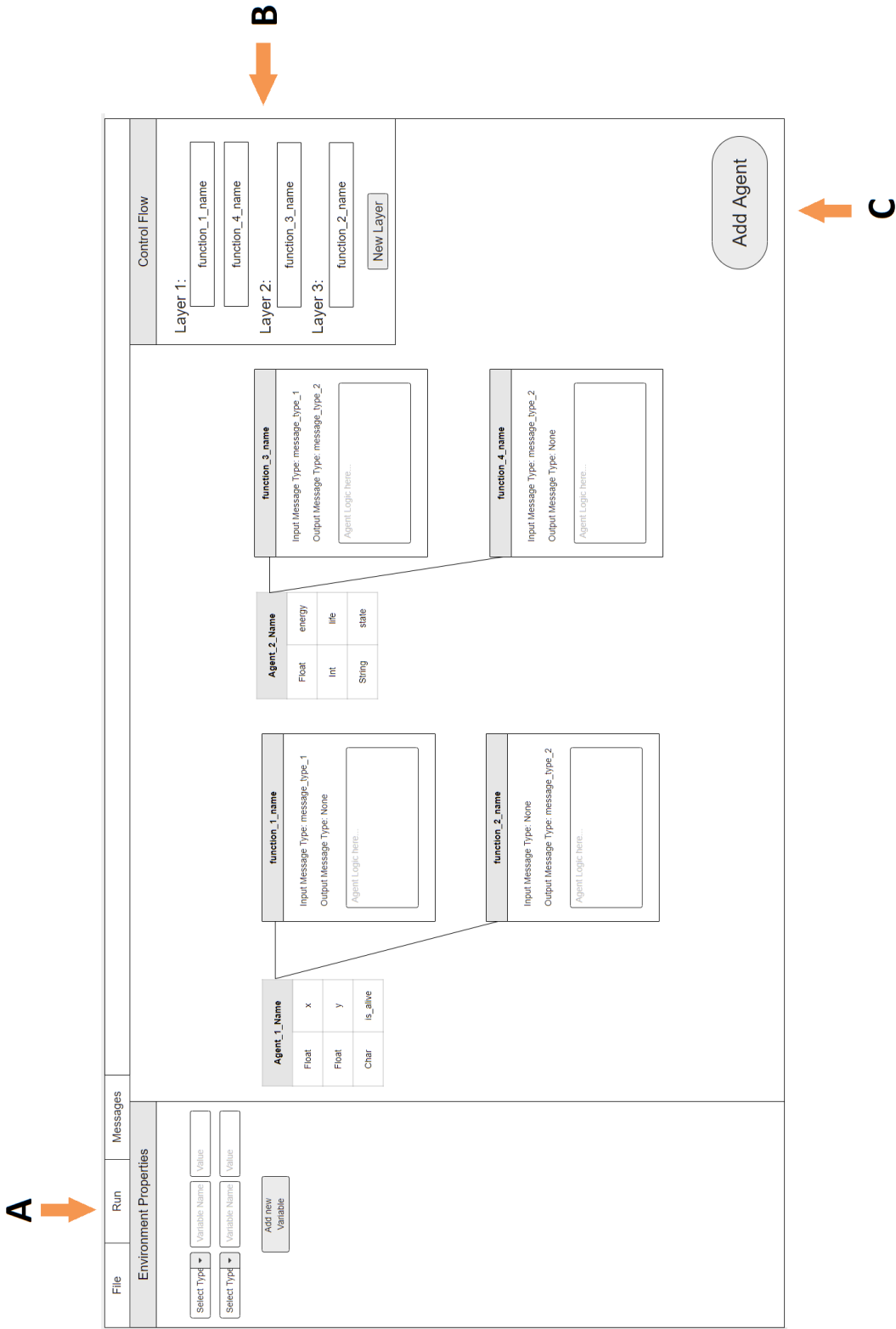


Figure 4.2: Proposed design for Main Screen

Messages

message\_name\_1

ID	id
Float	x
Float	y

DELETE

message\_name\_2

String	name
Int	value
Char	state

DELETE

Add New Message

Name:

Placeholder

Message Type:

Select

▼

Select

▼

Placeholder

Select

▼

Placeholder

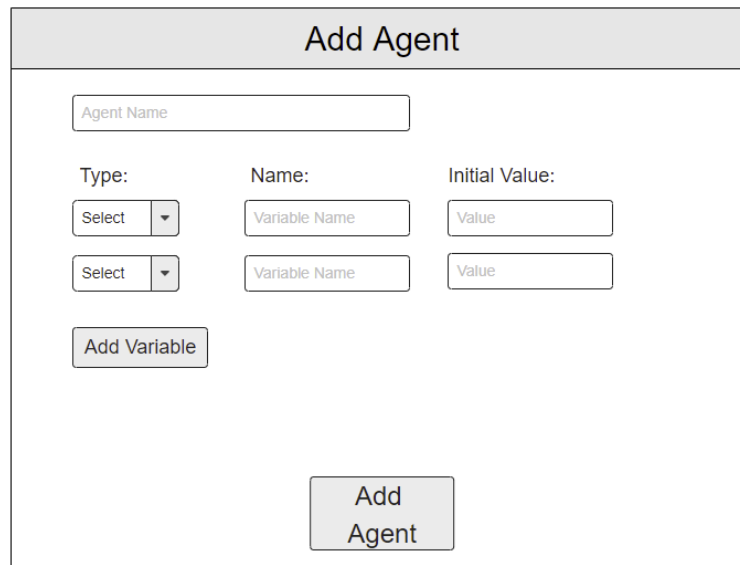
Add Variable

Add Message

Figure 4.3: Proposed design for Message Dialog

4.3.2 Message Dialog

Figure 4.3 shows the design for the dialog box where the user can see all the messages already created within the system and also allows them to create new ones. The container on the left hand side containing the preexisting messages will be scrollable as it wouldn't be possible to display all the messages in a complex simulation at one time. It will give the overview of the message, including its name along with the variable name and types with a delete button underneath each. It was decided to include this in the same dialog as where you can add new messages as it is possible that users may want o refer back to other messages when defining new ones. The rest of the dialog box will consist of a form, allowing the user to enter the required fields for new messages. This window purely focuses on completing part of requirement 1 from Table 3.2, as it is a component of defining the simulation as a whole.



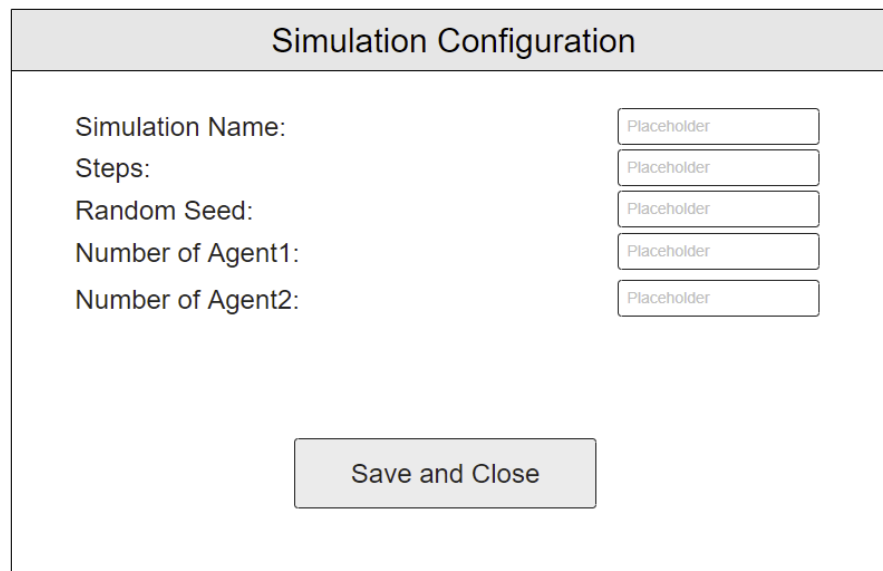
The diagram shows a dialog box titled "Add Agent". At the top is a text input field labeled "Agent Name". Below this, there are three columns of controls. The first column is labeled "Type:" and contains two "Select" buttons with downward arrows. The second column is labeled "Name:" and contains two text input fields, each labeled "Variable Name". The third column is labeled "Initial Value:" and contains two text input fields, each labeled "Value". Below the "Type:" column is an "Add Variable" button. At the bottom center of the dialog is a large "Add Agent" button.

Figure 4.4: Proposed design for Agent Creation Dialog

### 4.3.3 Agent Creation Dialog

To provide the ability to add new agents to the interface, there will be a dialog box shown in Figure 4.4 which will be accessible from the bottom right button on the Main Screen. The new agent's name and properties can be specified, however if the user wishes then they can not specify an initial value for one or any of the properties (FLAMEGPU2 interprets no initial value given as 0). In order to provide the most amount of usability to the user, in the initial value box it will be possible to enter either the name of an environment variable previously defined in the main window or a random function (as they exist in Python). This will give more flexibility to the user creating the model. Once finished, the user can click the "Add Agent" button to send the new agent data to the Main Screen to then create a new agent. This layout should be easy for the user to understand while still showing all relevant information to them. As with the previous dialog box, this works on meeting the requirement 1 of Table 3.2.





The image shows a 'Simulation Configuration' dialog box. It has a title bar at the top with the text 'Simulation Configuration'. Below the title bar, there are five labels on the left side: 'Simulation Name:', 'Steps:', 'Random Seed:', 'Number of Agent1:', and 'Number of Agent2:'. To the right of each label is a text input field containing the word 'Placeholder'. At the bottom center of the dialog box is a button labeled 'Save and Close'.

Figure 4.5: Proposed design for Configuration Dialog

#### 4.3.4 Configuration Dialog

Requirement 2 is achieved in the Configuration Dialog box (Figure 4.5) by specifying the simulation's execution such as the number of steps for which it runs, and initial agent populations. Similarly to the initial values in the agent creation dialog, population numbers can be set by Python random functions or predefined environment properties giving the user more flexibility and control of their simulation. A simple form containing the properties required to be defined including; step number, simulation name, random seed and all agent populations will be presented to the user to be filled in. As it is possible that a large number of agents could be in use in one model, if there be too many to fit in the dialog, a vertical scroll bar will appear to ensure the dialog does not stretch and become oversized.

## Chapter 5

# Implementation and Testing

### 5.1 Introduction

This section outlines how the program was made along with testing it to ensure it achieves the goal set out. It documents any changes that were realised during development, whether that be adding of features that had been overlooked or alterations to design.

### 5.2 Application Creation

#### 5.2.1 Creating the UI

To create the user interface for this program, an extension to Qt design framework (the one which will be used in this project) called Qt Designer was used. The decision for this was based on it being a What You See Is What You Get(WYSIWYG) approach rather than having to make the windows explicitly in code. Therefore making small alterations to how elements are positioned or look can be made without having to run the resulting Python script, reducing development time and reducing the chance of introducing bugs into the code.

Once the Main Window and dialog boxes were designed, they were saved through Qt Designer to a .ui file, a specifically formatted XML file. Then using the PyQt6 Python module, these files were compiled into Python source code files which then could be extended to include the code for the underlying logic and user interaction of the program.

Figure 5.1 shows the application's final interface after being used to create a basic model (unannotated screenshots of all windows and dialogs of the final program can be found in appendix E Figures E.1, E.2, E.3, E.4).

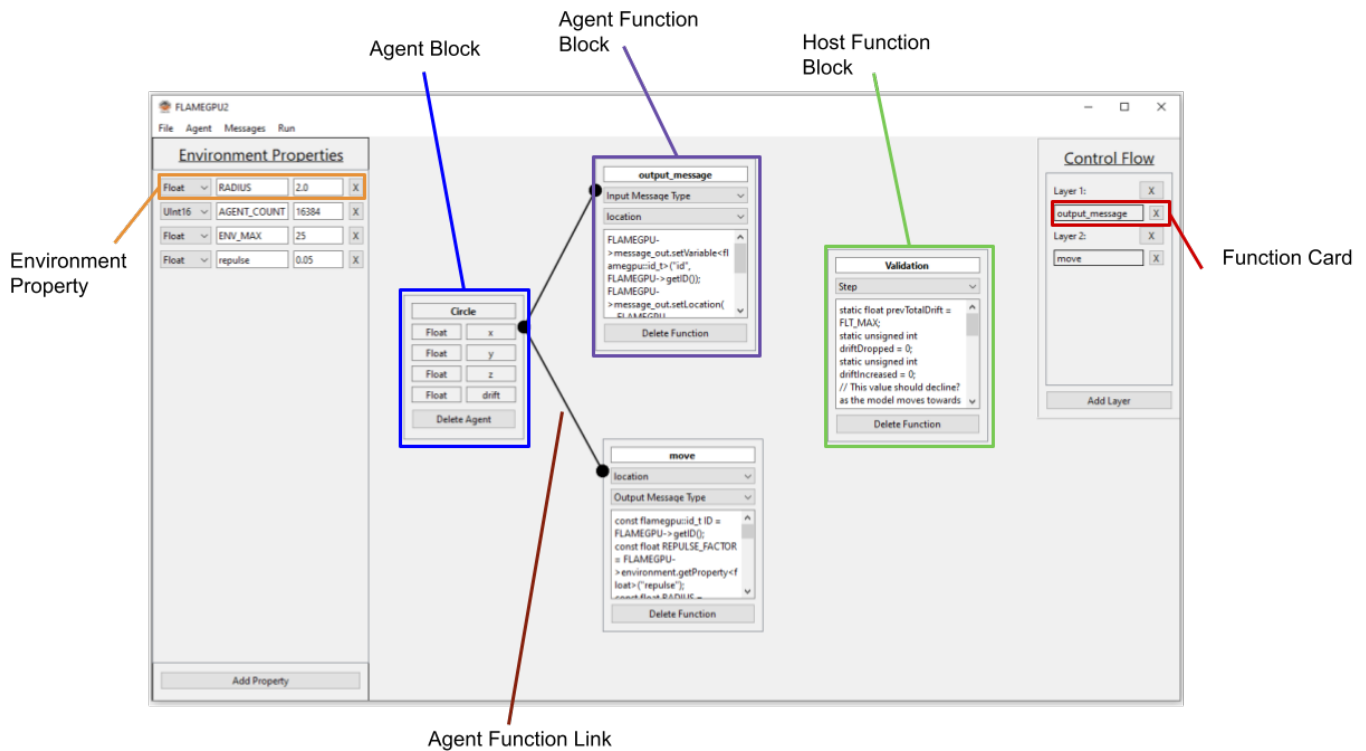


Figure 5.1: An annotated screenshot of the final application

### 5.2.2 Interface Behaviour

The way the user can interact with the elements on the screen is very important for this application and many features have been implemented in order to aid in the user experience.

In the main body of the main window, the agent (Figure 5.1 ‘Agent Block’) and function blocks (Figure 5.1 ‘Agent Function Block’) are freely movable by the user. This allows the user to visually structure their model as they wish, whether that be by agent or by execution order. As the function blocks can contain large amounts of code, the function block (both for the agents and the general system functions) are resizable by dragging the bottom right corner. This feature gives the benefit of being able to view the function code in full while also having the blocks at a reasonable size so you can see the rest of the defined model once finished.

A natural way to link agents with their corresponding functions was important to implement. It was decided to have users draw lines between the agent and function they wished to link (Figure 5.2 ‘Agent to Function Link’). The areas showing where to do this are denoted with a black circle. The function then only appears in the “Control Flow” box once it is linked to an agent, which gives the user the ability to have lots of functions defined in the application, linking and unlinking them to agents to test different behaviour functions. If more than one agent is linked to the same function, which may occur in models with different agents which

follow the same movement behaviour, the label in the control flow panel has the given agent name in brackets afterwards shown in Appendix E, Figure E.5. This not only makes it more obvious which agents it is acting on but also means two different agents can execute the same function at different times.

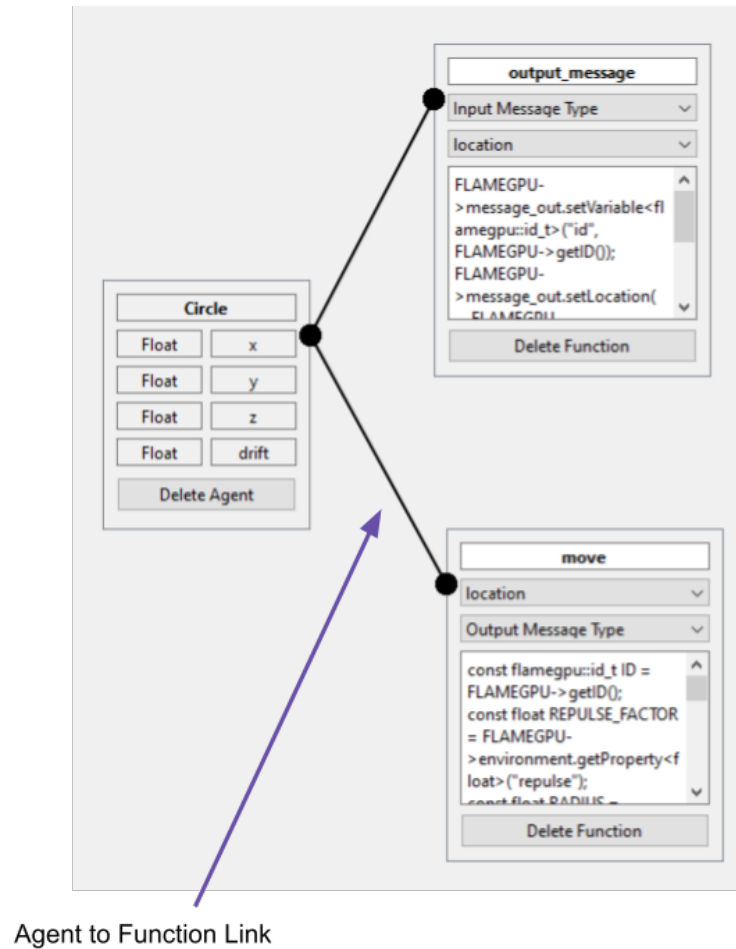


Figure 5.2: The link the user can make between agents and their functions

To give the user an easy way to edit agents that have already been defined, a double click feature was added, allowing the user to double click on any agent block to open the Agent Dialog populated with that agent's data. The user can then alter the agent's properties and confirm the dialog box, when the application will then update the agent, keeping all of its links to functions it had before.

In the vertically stacked Control Flow panel, all the linked functions are listed. To make it easy for the user to order the execution of the model as they wish, they can drag the function labels to where they want them to be. They can also add new layers by using the "Add Layer" button at the bottom.

Implementing a feature to save the model as it is defined in the application and then being able to load it again at a later date was imperative to allow for more complex models to be made. To be able to regain the exact state at a later date, all the data from the application from agents and their attributes to the execution order and block positions were stored in a JSON object. The decision to use a JSON object was taken as it is very readable in the code when it comes to dealing with the data stored within it. The JSON object that is created from the application state is saved to a .json file at the location and name of the user's choosing.

### 5.2.3 Validation

It is important to ensure all the data inputted by the user is valid so the application can compile the model into a FLAMEGPU2 Python script without any syntax errors. This includes variable names, variable values, function names and agent names. This had to be done robustly so that the user has the opportunity to rectify any errors rather than the application creating the FLAMEGPU2 script and attempting to run it at which point it would be unable to, forcing the user to potentially make changes directly to the script rather than within the interface. Whenever an input from the user is deemed invalid, the current process is halted and an error message box appears telling the user an error had been found, and where, allowing them to change it.

All names defined by the user are validated in the same way whether it be variable, agent or function names. The validation ensures they only contain alphanumeric characters with the only exception being an underscore therefore meaning they are valid names in Python. The check also rejects the name if its first character is a digit.

Variable values that are input run through multiple checks to confirm whether they are valid or not. There are four main areas where the user must give numerical values in the process of defining their model, these are; environment property values, agent variable and population values, message variable values and configuration values. In all four of these areas, the value entered is checked that it matches the type that is either defined by the user (e.g an agent variable) or that is required by the system (e.g camera position).

An additional feature is present in all these four areas apart from the configuration values where the user can choose to enter a random function recognised by Python as opposed to a literal value. For example if the model were to consist of an agent which had a property of "energy" and the user wished for each instance of that agent to have an integer "energy" value of between 15 and 20, they would be able to describe this by entering "random.randint(15, 20)". This gives the user much more flexibility in the types of models they can define. When it comes to validating these values, the function entered is evaluated by the application to ensure its type matches the one required/defined.

To give even more options to the user in how they create their model, the application allows them to enter variable names that have been predefined in the environment properties section of the Main Screen in both the Agent and Message Dialogs. This feature makes it easier when making changes to the model as rather than having duplicate values that would

need to be changed, it can be in one place with multiple other references to it. This can also be used in conjunction with the random function feature, where an environment variable is defined by some random function, and then an agent's property is given as the environment property. This allows the introduction of randomness for characteristics of the system while still allowing other parts of the model to use the value.

#### 5.2.4 Code Generation

To generate the script after the user has defined the model, a code generation class very similar to the proposed in Chapter 2, Algorithm 1 was used. The main difference is the additional support for either writing single lines or being able to give a list of lines of code to add. The inclusion of this feature reduced the amount of code required when generating the FLAMEGPU2 script.

---

**Algorithm 2** The class used to generate the Python FLAMEGPU2 script

---

```
class CodeGen:

    def __init__(self, tab="\t"):
        self.code = []
        self.tab = tab
        self.level = 0

    def write(self, lines, indent = 0):
        if isinstance(lines, list):
            for line in lines:
                self.code.append(self.tab * self.level + line)
        else:
            self.code.append(self.tab * self.level + lines)
        self.indent(indent)

    def indent(self, indent = 1):
        if self.level + indent >= 0:
            self.level += indent
        else:
            print("Indentation error")

    def save(self, filename):
        with open(filename, "w") as outfile:
            outfile.write("\n".join(self.code))
```

---

An instance of the class shown in Algorithm 2 was used to generate all the code required

for the FLAMEGPU2 script. This was a very useful approach as the instance was then easily used in loops where a large number of variables needed to be added to the model.

## 5.3 Changes Made During Development

### 5.3.1 Visual changes

Some small visual changes were made to the interface in comparison to the mock-ups in the design section as during the implementation, it was clear some widgets provided a better user experience and prevented overcrowding being in other places. The visual changes are:

- The add agent button was moved from the bottom right corner to a dropdown under the “Agent” menu item at the top (seen in Figure E.1)
- Agent population is given in the Agent Dialog rather than the configuration dialog (seen in Figure E.2)
- Agent visualisation option (such as mesh, colour, scale etc) is given in the Agent Dialog (seen in Figure E.2)

### 5.3.2 Additional Features

During development, some additional features in FLAMEGPU2 were found that required to be implemented to ensure a full model was possible to be defined in the framework.

One of these features is the inclusion of host functions. Host functions are blocks of code that are run by the model but not directly by a specific agent. They take one of four types, with the type defining when the function executes. These are “Init” which runs at the very start of the model; “Step” which runs after every iteration; “Exit” which executes at the very end of the model; and “Host-Layer” which the user specifies at which point in the step it executes just like a regular agent function. These were implemented into the application in a very similar way to the standard agent functions, as a movable block contained within the Main Screen.

Another addition that was previously overlooked was the need for certain default variables and values for certain types of messages. For example in a SpatialMessage, an interaction radius and a minimum value (whether that be in 2 or 3 dimensions) is needed (shown in Figure 5.3). Therefore to account for this, extra entries appear dynamically in the Message Dialog depending on the type of message that is being defined.

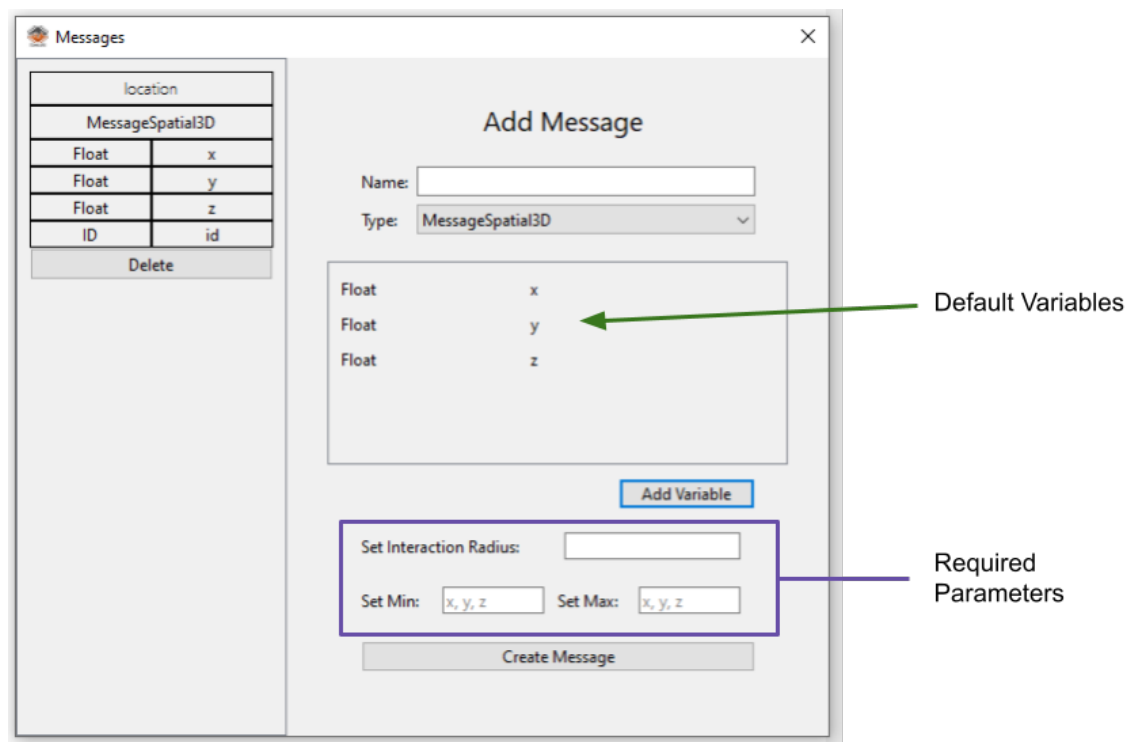


Figure 5.3: Message Dialog showing default values for a SpatialMessage3D

## 5.4 Testing

Testing is a valuable part of any project. Specifically in this situation not only is it important to ensure that all the component parts of the application work properly and as expected, but also that the application as a whole can work together to give the desired experience. Therefore to ensure thorough testing and be able to test how robust the system is, two types of testing took place, integration and end-to-end testing. The former tested individual parts, accessing their behaviour and the latter runs through a scenario expected from an end user as how they would use it.

### 5.4.1 Integration Testing

These tests are designed to test whether the functionality implemented into the system works as intended. It was carried out manually as opposed to automating it. While it is possible to define these tests and carry them out automatically through a part of the PyQt6 library, it was decided to do them manually due to time constraints. If there were a longer period of time for testing, then the time to write the tests to execute automatically would be warranted, however this does not change the efficacy of the testing as it still tests every element. This section of testing tests everything from validation and addition of new elements to visual



components and saving. It is split up into four sections, one for each different window, namely the Main Screen, Agent Dialog, Message Dialog and Configuration Dialog.

No.	Description	Success?
1	Does it open	Yes
2	Add an environment property	Yes
3	Delete an environment property	Yes
4	Move a block	Yes
5	Resize a function block	Yes
6	Enter properties for a function block	Yes
7	Link an agent to a function	Yes
8	Link multiple agents to one function	Yes
9	Link one agent to multiple functions	Yes
10	Reorder functions in control flow panel	Yes
11	Add a layer to control flow panel	Yes
12	Remove a layer from control flow panel	Yes
13	Remove an agent function link by deleting the function in the control flow panel	Yes
14	Delete an agent block	Yes
15	Delete an agent function block	Yes
15	Delete a host function block	Yes
16	Save the application to a .json file	Yes
17	Load a .json file into the application	Yes
18	Launch simulation	Yes
19	Visualisation of simulation	Yes

Table 5.1: Integration testing for Main Window

No.	Description	Success?
1	Does it open	Yes
2	Accept an environment property as value	Yes
3	Accept a Python random function as value	Yes
4	Reject incorrect value type for variable type defined	Yes
5	Add new variable	Yes
6	Close and create agent block	Yes

Table 5.2: Integration testing for Agent Dialog

No.	Description	Success?
1	Does it open	Yes
2	Do default properties appear when message type is changed	Yes
3	Are values for default properties of incorrect type rejected	Yes
4	Accept an environment property as value	Yes
5	Accept a Python random function as value	Yes
6	Add a variable	Yes
7	Create a message	Yes
8	Pre-exisitng messages appear on left hand column	Yes

Table 5.3: Integration testing for Message Dialog

No.	Description	Success?
1	Does it open	Yes
2	Are incorrectly formatted values for camera position and direction rejected	Yes
3	Closes and saves	Yes
4	Rejects invalid simulation name	Yes

Table 5.4: Integration testing for Configuration Dialog

As Tables 5.1, 5.2, 5.3 and 5.4 show, the application performed very well in this phase of testing. All functionality reacted as it was expected to and nothing caused the program to crash or act strangely.

#### 5.4.2 End-To-End Testing

This second area of testing replicated how it is expected that an end user would use the system to see if it performs in a suitable manner overall. To do this testing, a replication of an example given on the FLAMEGPU2 GitHub (video of simulation is available here <https://www.youtube.com/watch?v=ZedroqmOaHU>) where a large number of agent congregate into local spheres from random starting locations. The reasoning for choosing this model is it is not too complex such that it would take too long to implement into the application but it still uses all the component parts of the application including; agents, environment properties, agent functions, host functions, reordering function execution, using environment properties to define variable values and using Python random functions to define variable values. The full action by action list of how this testing was carried out is given in table F.1.

The resulting simulation from this testing is shown in Figures 5.4, 5.5, 5.6, 5.7.

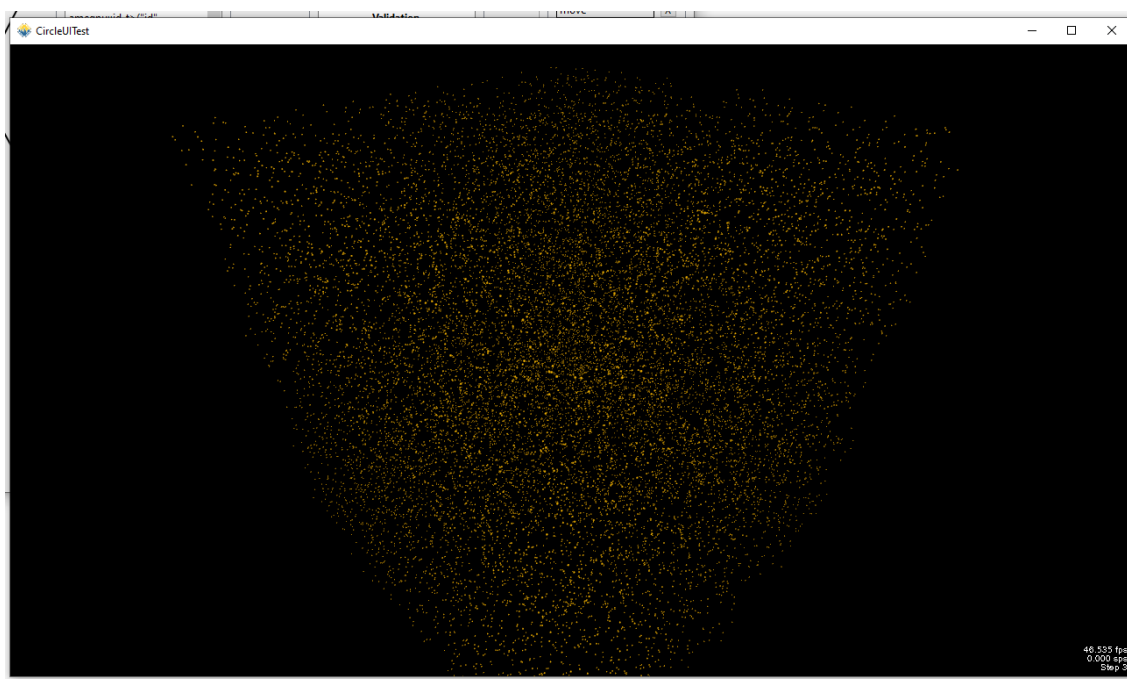


Figure 5.4: Simulation at the initial stage

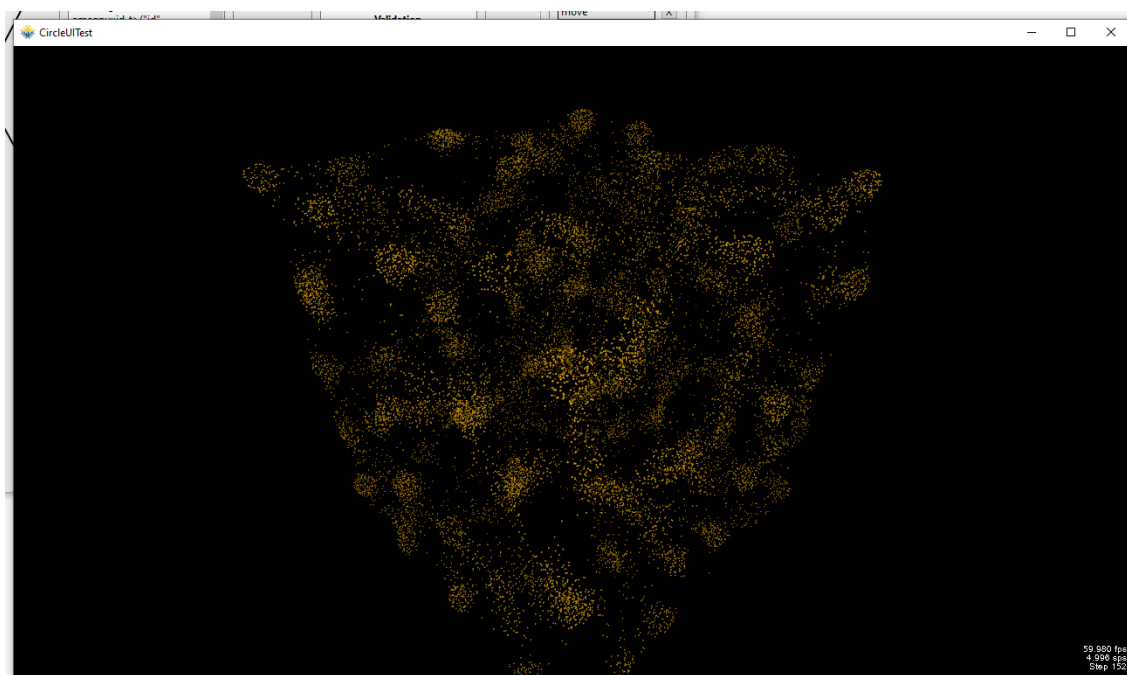


Figure 5.5: Simulation after agents start to congregate

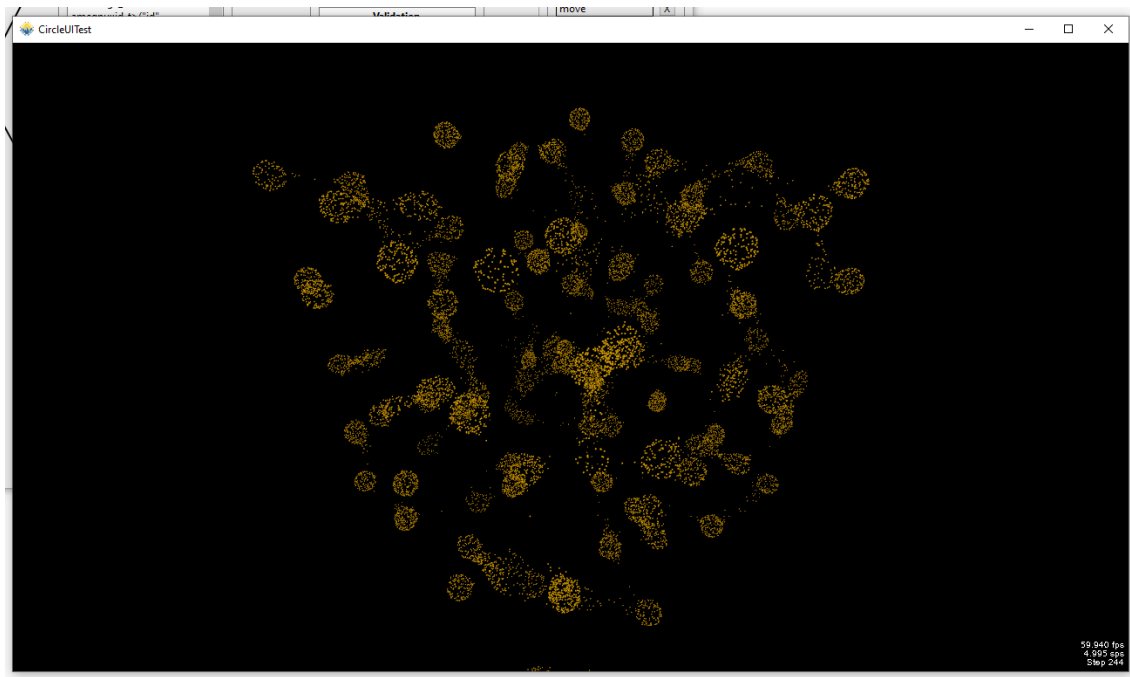


Figure 5.6: Simulation after agents form first spheres

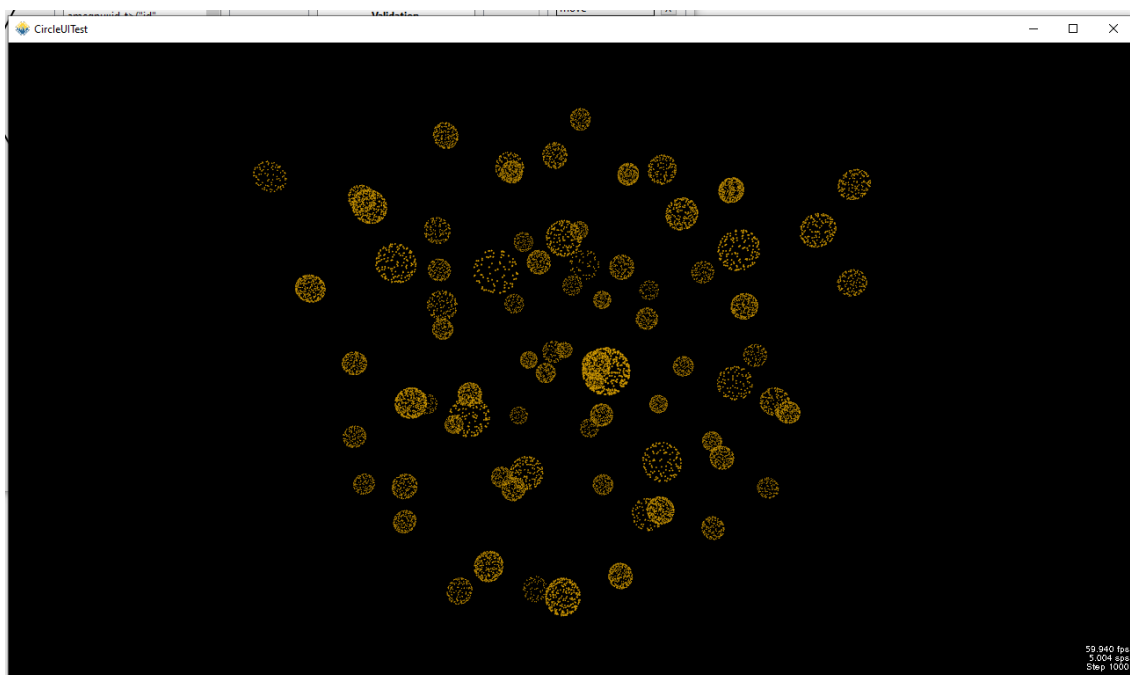


Figure 5.7: Final stage of simulation where agents find equilibrium

This test performed as it was expected to, being able to define and run the simulation without any errors occurring. It ran without any major lags, with it only taking approximately

2 seconds to generate the FLAMEGPU2 script and run the simulation from when the “Launch” button was pressed. This shows the application is robust and all its features were implemented successfully.

## Chapter 6

# Results and Discussion

### 6.1 Outcome

This project has resulted in an application capable of defining agent-based models in the framework FLAMEGPU2 purely through interaction with a GUI with no knowledge of programming or the structure of FLAMEGPU2 scripts needed. This reduces the knowledge required for someone to use this framework and therefore opens up the benefits of it to a much larger number of people.

The core of functionality in the FLAMEGPU2 framework has been implemented, allowing users to define full models. Users can define agents and their properties including their initial population and visualisation options (if they want to visualise them, colour, mesh and scale). Agent functions can be created and can be linked to any number of different agents (zero or more) with the execution of the function for each distinct agent type being able to be specified independently. New messages for function inputs/outputs can be established, ensuring that the user gives values for required properties of certain message types such as an interaction radius for a MessageSpatial3D. Environment properties can be defined and used throughout the application in places such as initial agent values and message properties. Finally, some configuration settings for the simulation can be set out including the simulation name, how many steps it will execute for, the speed of execution and camera position/directions. As FLAMEGPU2 is a large and actively developed framework, some more niche features resulted in being out of scope of this project however this would not affect the majority of users.

### 6.2 Evaluation of Application

#### 6.2.1 Meeting of Requirements

In the Requirements and Analysis section of this report, 12 requirements were set out (Table 3.2) as guides for the features of the program and their priority. By referring back to this, we can evaluate to what extent the final application met the initial aims of the project. An extension of the original requirements table shown in Table 6.1, showing to what degree each

was met.

Number	Requirement	Priority	Achieved?
1	Define new ABM simulation elements including agents, messages, agent functions and environment properties through the GUI	M	Yes
2	Set simulation configuration such as number of steps it will run for and initial populations of agents	M	Yes
3	Specify execution order of agent functions by organising into layers	M	Yes
4	Run the simulation from the GUI	M	Yes
5	View a graph of agent populations over time after the simulation has executed	D	No
6	View a visualisation of the model as it executes	D	Yes
7	Generate and save a standalone Python source code file of the simulation	D	Yes
8	Define agent populations by attaching an XML file	D	No
9	Load pre-existing FLAMEGPU2 scripts into the GUI to view them	O	No
10	Change attributes of the visualisation (e.g agent models, colours, etc)	O	Yes
11	Define control flow through a dependency graph rather than layers	O	No
12	View any agent attribute over time in a graph	O	No

Table 6.1: Requirements Achieved

Table 6.1 demonstrates that over half of the requirements were met, with a percentage skew towards the mandatory and desired requirements. The requirements that describe the base system (requirements 1, 2, 3 and 4) were all met, meaning that the application was successful in being able to be used to define an entire ABM within the FLAMEGPU2 framework and execute it. The reason for requirements 6 and 7 being implemented out of all of the desired priority requirements was that they provided the most useful additional functionality and would more likely to be used in the real world than requirements 5 and 8.

The reason for the five requirements not being implemented was due to time constraints in the development phase of this project. With the exception of requirement 9, none of the unachieved requirements would be difficult to implement from a technical standpoint. Adding the ability to load in pre-existing FLAMEGPU2 scripts into the application with the ability to then make alterations to them would have posed a challenge in the way it was implemented however not to a level that would have been out of the ability of this project. With improved time management or the existence of a longer development phase, the requirement list would have been implemented in full.

### 6.2.2 Evaluation

As a whole the application provides the user with the tools for creating the ABM they wish in a very straightforward and intuitive way, with all information being no more than 2 clicks away from the Main Screen. Therefore it achieves its goal set out in the Requirements and Analysis section of increasing the usability of FLAMEGPU2.

However while the application created for this project is a large stride in making FLAMEGPU2 more accessible to people who have very little or no programming knowledge, it does have a shortcoming. Due to agent and host functions still requiring full C++ and Python code respectively for defining what actually happens, it means that someone who cannot code can't create a model entirely on their own. They would still require some extra help or to learn how to code themselves. Despite this, during the end-to-end testing, only 60 lines of code were required to be input. This is in contrast to the total 151 lines of code that the application generated in the standalone FLAMEGPU2 script once the simulation was executed showing that even in a relatively simple model, the amount of code required to define it is reduced by roughly 60%. A solution to this issue of still requiring an amount of code is out of scope of this project and would require a significant amount of time to produce a succinct solution that doesn't limit what can be created with it. Nevertheless this does not make the solution created in this project obsolete, it still gives more control to researchers or others who are not proficient in coding to define the main aspects of the model such as agents and their attributes and the general execution order of the program.

While it is hard to follow the WYSIWYG design principle exactly for this type of application, largely it is met. The user interface represents the way the model is structured exactly with none of the information represented in such a way that obfuscates it. This makes it easy to convey how the model is structured and its behaviour to another person while using the application, providing benefit for users who act as a part of a team.

## 6.3 Further Work

While the application created for this project was successful in creating an effective GUI for FLAMEGPU2, some improvements and changes could definitely be made to make it an even more useful tool.

### 6.3.1 Extensions to This Project

Some of the changes that would improve the application could be made directly to it rather than starting development of a whole new application.

The main area would be to fully implement all the requirements that were listed previously. By adding requirements 4 and 12, the user would be given more information about the actions during the simulation and therefore be able to get more usage out of a model. If the option of being able to define a model's initial state (such as populations and attribute values) via an XML file, it could potentially save users time and make it easier for changes to be made.



The ability of being able to load in a FLAMEGPU2 script into the UI to view it and make changes this way would offer much more flexibility to the user and mean the GUI could be used in projects which already have models created in FLAMEGPU2. The final requirement that wasn't met was the ability to define the execution order of the simulation through a dependency graph to bring it up to date with FLAMEGPU2 as opposed to the first iteration. This could use a similar system to the current agent/function linking where blocks exist of the screen and then the user can drag lines between them. This could be visually in the same place as the agents and functions but on a separate tab giving the user the ability to swap between the two.

A further extension to the application could be in the form of submodels. Submodels in FLAMEGPU2 are when a more basic model is nested within a parent model, with the submodel being called per step of the simulation. The point in each step when the submodel is executed is defined in the same way as the agent functions. The inclusion of this would allow more complex models to be created with the application.

Being able to give more control as to how the simulation is visualised would be beneficial, specifically changing an agents colour based on its properties. This feature is supported within FLAMEGPU2 but was out of scope of this project however the addition of it would mean visualisations would be able to provide much more information on the state of the simulation. There are many examples where this feature would be useful such as an agent's colour changing based on if they were infectious (in a simulation representing disease spread) or just to differentiate between types of agents (if their model meshes are the same).

If this project were to be maintained long term, the implementation of fully automated unit testing would be required. Writing tests that can execute automatically to evaluate the functionality of the UI would be very important, saving time as they would not need to be carried out manually each time. It would ensure that when changes were made to the program, they did not inadvertently cause issues to functionality elsewhere.

### 6.3.2 Ideas For New Projects

Since completing this project, many other potential projects related to this have come to mind as a different way of approaching this issue. Therefore this project could methodically lead the way into a whole separate project building on what has been learnt here.

A new way of creating this application could be an online based GUI editor. The functionality of this online version could be largely similar in how the user may define the aspects of their model but with the added benefit of it being accessible from any computer with an internet connection and without the need for a dedicated CUDA enabled GPU and a Python installation. When it then comes time to run the simulation, two options could be given; if the user is using a computer with the suitable hardware and software requirements then they could generate the script to then be downloaded and run locally. Alternatively the script could automatically be transferred over to a Google Collab file where using the hardware acceleration of that service the simulation could be run in the cloud. One of the

example projects<sup>1</sup> provided on the FLAMEGPU2 GitHub page uses Google Collab in this way, giving the inspiration for this new approach and the knowledge that it would work. This would further open the usability of the application as people with laptops or weaker desktop PCs would still be able to use it. As it would be a website based application, it would largely have to be re-written due to the vastly different technologies used to build websites as opposed to desktop applications like the one created in this project and therefore would warrant a standalone project.

---

<sup>1</sup>Link to demo project on Google Collab <https://flamegpu.com/try/>

## Chapter 7

# Conclusion

The goal of this project was to increase the usability of FLAMEGPU2 which was achieved by creating a UI to provide people with a way of defining ABMs within the framework visually and without the need to program any aspect of it. With FLAMEGPU2 being a relatively new framework, its user base is quite small and therefore by creating this application the user base can be extended to people without advanced coding knowledge.

A variety of pre-existing GUI implementations for ABM frameworks were investigated in Chapter 2, which provided evidence for the importance of user oriented design. While these solutions were effective in providing a visual way for users to define models, as none of them were built on the FLAMEGPU2 framework, they did not impart the performance benefits that inherently come with the framework running models directly on the computer's GPU.

During the design stage, mock ups were created of each window and dialog based on the requirements set out for a successful system. The design was done in such a way as to give as much information to the user as easily as possible without overwhelming them and creating confusion. The structure of the main screen of the application and how the user links agents and functions took inspiration from the statecharts that were seen in the literature review. These statecharts created a visual way of easily understanding the structure of the model, which was wanted to be imitated to increase usability in this project. All the details about the model that they define is never any further than two clicks away making navigating the UI easy.

The application was written in Python version 3.8 and utilised a GUI building module called PyQt6 for the visual aspects of it. Multiple options for how to create the GUI were explored in Chapter 2, with PyQt6 being chosen due to its wide range of features and its visual editor it provides. Aesthetically the application did not change much from the design phase of the project with only a few placements of elements changing. Integration and end-to-end testing was done to assess the performance of the program and to simulate how an end user may use the system.

Largely, this project was a success by creating a tool for users to be able to define and run a model within the FLAMEGPU2 framework without coding. In the testing conducted in this project, it was proven that the amount of code required from the user to define a

model can be reduced by as much as 60%, producing a vast increase in usability for users. While some amounts code are still required for function behaviour, the amount of information needed for a user to create a whole model using the system is considerably less than if they were to entirely write the model themselves as they do not need to have any knowledge of the structure of FLAMEGPU2 scripts, reducing the possibility of syntax errors massively. A few requirements were not met however these were not key parts functionally as they were not relevant to defining the model. Therefore overall the aim of this project to increase the usability of the framework is met.

Multiple extensions or new project could come from the completion of this one; the most impactful being an online version of this application that makes use of Google Collab and its hardware acceleration feature. By creating this, it would allow people without access to computers with dedicated CUDA enabled GPUs to get the performance benefits that FLAMEGPU2 offers. Another advantage would be that a model could be worked on from anywhere in the world as long as they have an internet connection making collaborative efforts much easier. This would further increase the usability of FLAMEGPU2, bringing access to it to more people.

In summary, a GUI has been created in Python for the FLAMEGPU2 framework which allows users to define ABMs without the need to have knowledge of how it structures scripts, consequently increasing the usability of the framework. The UI provides an intuitive way for people without significant coding knowledge to create models and gain the benefits of using high performance ABMs in their work, therefore achieving the aim of this project.

# Bibliography

- [1] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008. General-Purpose Processing using Graphics Processing Units. URL: <https://www.sciencedirect.com/science/article/pii/S0743731508000932>.
- [2] Robert Clay, Jonathan A. Ward, Patricia Ternes, Le-Minh Kieu, and Nick Malleson. Real-time agent-based crowd simulation with the reversible jump unscented kalman filter. *Simulation Modelling Practice and Theory*, 113:102386, 2021. URL: <https://www.sciencedirect.com/science/article/pii/S1569190X21000939>.
- [3] Robin Dunn. wxpython 4.1.1. URL: <https://pypi.org/project/wxPython/>.
- [4] Jeff Dyck, David Pinelle, Barry AT Brown, and Carl Gutwin. Learning from games: Hci design innovations in entertainment software. In *Graphics interface*, volume 2003, pages 237–246. Citeseer, 2003.
- [5] Wilbert O Galitz. *The essential guide to user interface design: an introduction to GUI design principles and techniques*. John Wiley & Sons, 2007.
- [6] Ralph F Grove and Eray Ozkan. The mvc-web design pattern. In *International Conference on Web Information Systems and Technologies*, volume 2, pages 127–130. SCITEPRESS, 2011.
- [7] Tadashi Kurata, Hiroshi Deguchi, and Manabu Ichikawa. Gui for agent based modeling. In Hideki Takayasu, Nobuyasu Ito, Itsuki Noda, and Misako Takayasu, editors, *Proceedings of the International Conference on Social Modeling and Simulation, plus Econophysics Colloquium 2014*, pages 275–286, Cham, 2015. Springer International Publishing. URL: [https://link.springer.com/chapter/10.1007/978-3-319-20591-5\\_25](https://link.springer.com/chapter/10.1007/978-3-319-20591-5_25).
- [8] River Computing Limited. Pyqt6 6.3.0. URL: <https://pypi.org/project/PyQt6/>.
- [9] Fredrik Lundh. An introduction to tkinter. URL: [www.pythonware.com/library/tkinter/introduction/index.htm](http://www.pythonware.com/library/tkinter/introduction/index.htm), 1999.

- [10] Adam J. McLane, Christina Semeniuk, Gregory J. McDermid, and Danielle J. Marceau. The role of agent-based models in wildlife ecology and management. *Ecological Modelling*, 222(8):1544–1556, 2011. URL: <https://www.sciencedirect.com/science/article/pii/S0304380011000524>, doi:<https://doi.org/10.1016/j.ecolmodel.2011.01.020>.
- [11] Josef Noll and Carlos Westphall. *MOBILITY 2015 - The Fifth International Conference on Mobile Services, Resources, and Users*. 06 2015. doi:10.13140/RG.2.1.1901.4165.
- [12] Michael J North, Nicholson T Collier, Jonathan Ozik, Eric R Tatara, Charles M Macal, Mark Bragen, and Pam Sydelko. Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling*, 1(1), mar 2013. URL: <https://link.springer.com/article/10.1186/2194-3206-1-3#citeas>, doi:10.1186/2194-3206-1-3.
- [13] Paul Richmond, Robert Chisholm, Peter Heywood, Matthew Leach, and Mozhgan Kabiri Chimeh. Flame gpu. 2021. doi:10.5281/ZENODO.5428984.
- [14] Armin Ronacher. Jinja2 3.1.2. URL: <https://pypi.org/project/Jinja2/>.
- [15] Kivy Team. Kivy 2.1.0. URL: <https://pypi.org/project/Kivy/>.

# Appendices

## Appendix A

# Tkinter Sample Code

```
from tkinter import *

root = Tk()
root.title("Example Window")
root.geometry("300x300")

test_label = Label(root, text = "Example Label")
test_button = Button(root, text = "Example Button")
testTextBox = Entry(root, text = "Example Text Box")

test_label.pack()
test_button.pack()
testTextBox.pack()

root.mainloop()
```



## Appendix B

# PyQt6 Sample Code

```
from PyQt6.QtWidgets import *
from PyQt6.QtCore import *
import sys

root = QApplication(sys.argv)

window = QWidget()
window.setWindowTitle("Example Window")
window.setFixedSize(QSize(300, 300))

button = QPushButton(window)
button.setText("Example Button")

label = QLabel(window)
label.setText("Example Label")

textBox = QLineEdit(window)

box = QVBoxLayout()
box.addWidget(button)
box.addWidget(label)
box.addWidget(textBox)
box.addStretch(1)

window.setLayout(box)
window.show()

root.exec()
```

## Appendix C

# Kivy Sample Code

```
import kivy
from kivy.app import App
from kivy.core.window import Window
from kivy.uix.widget import Widget
from kivy.uix.button import Button
from kivy.uix.label import Label
from kivy.uix.textinput import TextInput
```

```
Window.size = (300,300)
```

```
class MyWindow(Widget):
    pass
```

```
class ExampleWindow(App):
    def build(self):
        return MyWindow()
```

```
ExampleWindow().run()
```

```
<MyWindow>:
    GridLayout:
        rows: 5
        size: root.width, root.height

        Label:
            text: "Example Label"

        Button:
            text: "Example Button"
```

```
TextInput:
    text: "Example Text Box"
```

## Appendix D

# WxPython Sample Code

```
import wx

app = wx.App()

frame = wx.Frame(None, title='Example Window', size = (300,300))

label = wx.StaticText(frame, 1, "Example Label")
button = wx.Button(frame, 1, "Example Button")
textBox = wx.TextCtrl(frame, 1, "Example Text Box")

vbox = wx.BoxSizer(wx.VERTICAL)

hbox1 = wx.BoxSizer(wx.HORIZONTAL)
hbox1.Add(label, 1, wx.EXPAND)

hbox2 = wx.BoxSizer(wx.HORIZONTAL)
hbox2.Add(button, 1, wx.EXPAND)

hbox3 = wx.BoxSizer(wx.HORIZONTAL)
hbox3.Add(textBox, 1,wx.EXPAND|wx.ALIGN_LEFT|wx.ALL,5)

vbox.Add(hbox1)
vbox.Add(hbox2)
vbox.Add(hbox3)

frame.SetSizer(vbox)

frame.Show()
```

```
app.MainLoop()
```

# Appendix E

## Application Screenshots

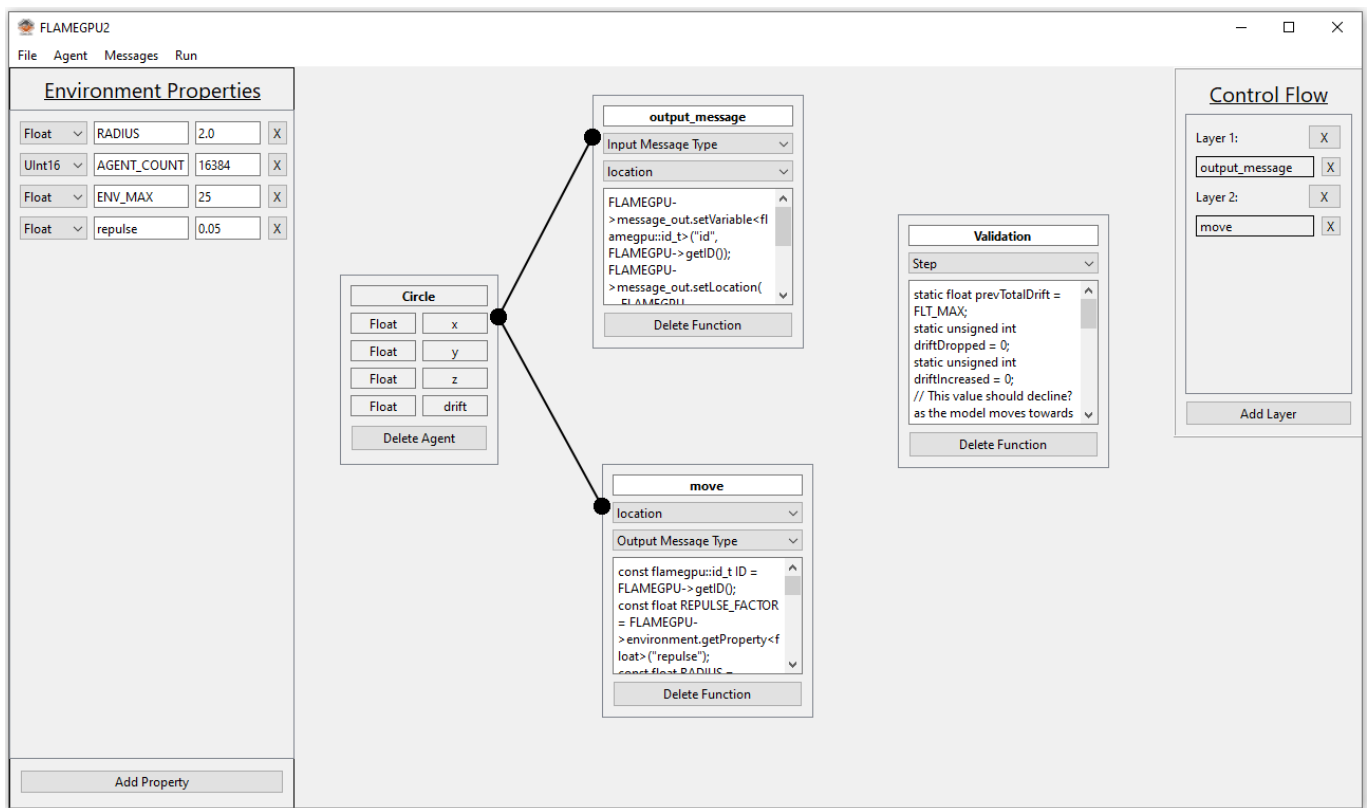
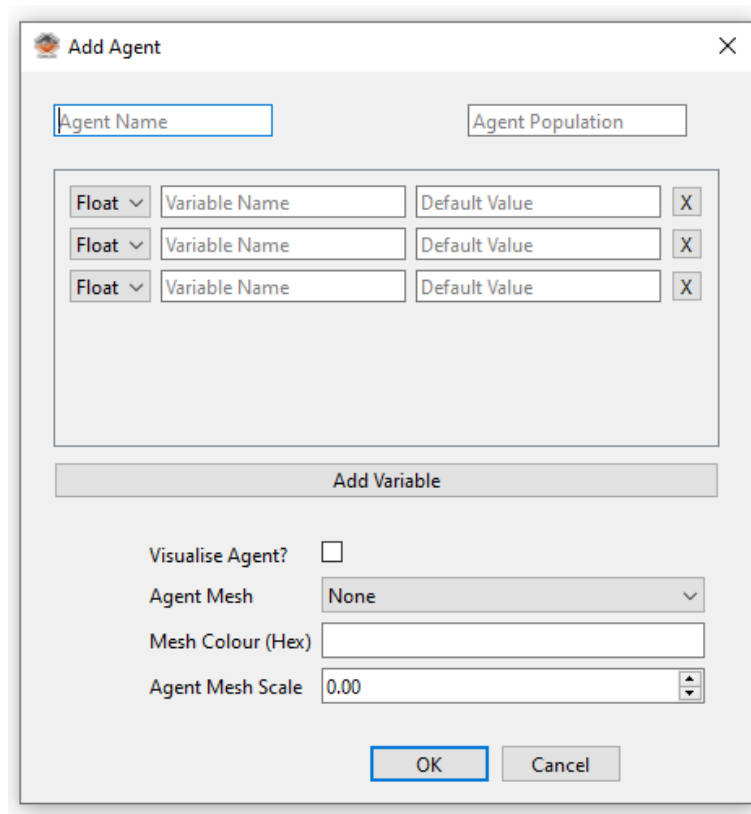


Figure E.1: Screenshot of the final application with a basic model created using it



The screenshot shows a software dialog box titled "Add Agent" with a close button (X) in the top right corner. The dialog contains the following elements:

- Two text input fields at the top: "Agent Name" and "Agent Population".
- A central container with three rows of variable definitions. Each row consists of:
  - A dropdown menu set to "Float".
  - A text input field labeled "Variable Name" containing the placeholder text "Variable Name".
  - A text input field labeled "Default Value" containing the placeholder text "Default Value".
  - A small button labeled "X" to the right of the "Default Value" field.
- A button labeled "Add Variable" located below the variable list.
- A checkbox labeled "Visualise Agent?" which is currently unchecked.
- A dropdown menu labeled "Agent Mesh" with "None" selected.
- A text input field labeled "Mesh Colour (Hex)".
- A spinner control labeled "Agent Mesh Scale" with the value "0.00".
- "OK" and "Cancel" buttons at the bottom right.

Figure E.2: Final Agent Dialog showing 3 sample variables added

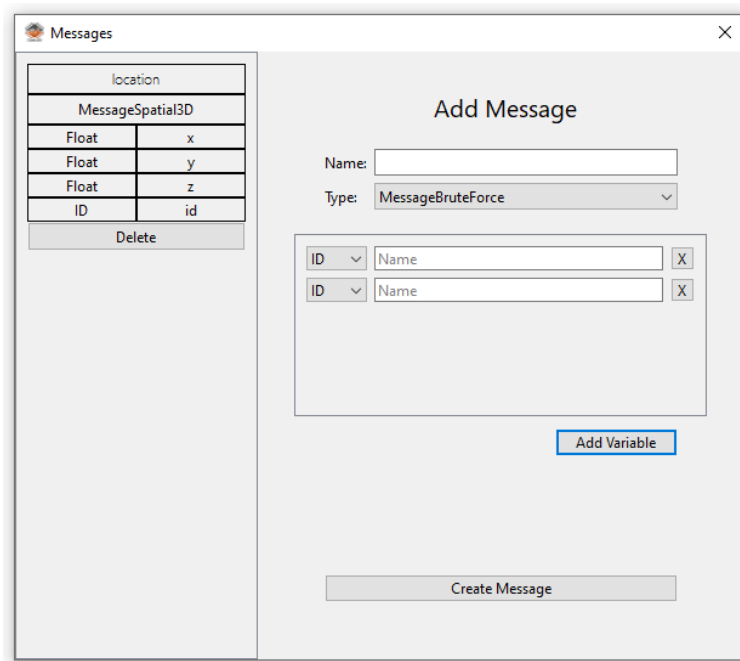


Figure E.3: Final Message Dialog showing one message already created and 2 sample variables added

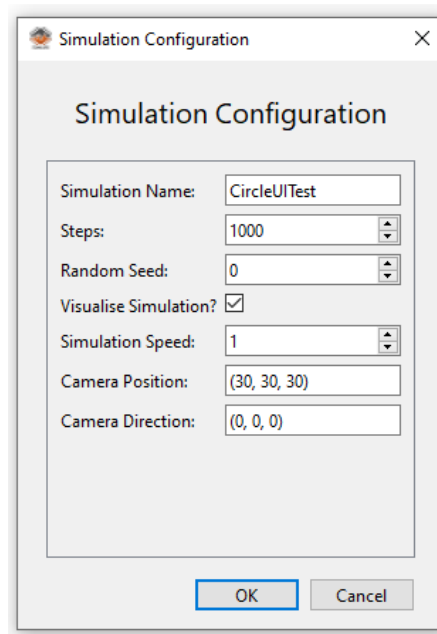


Figure E.4: Final Configuration Dialog to setup how the simulation will run



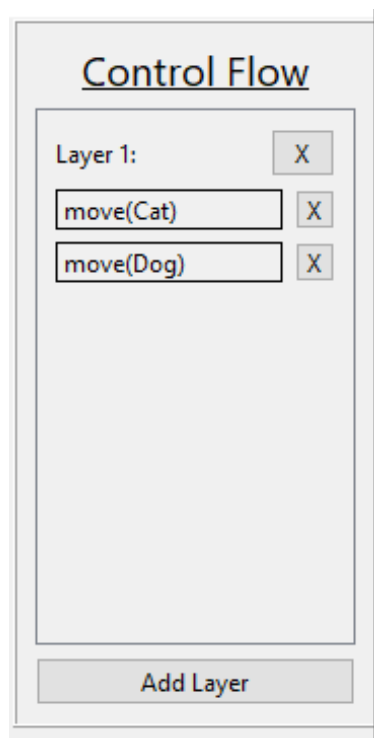


Figure E.5: The control flow panel when multiple agents are linked to the same function

## Appendix F

### Testing

No.	Description	Action	Input Data
1	Add new environment property	Click “Add Property”; Enter values into fields	“Float”; “RADIUS”; “2.0”
2	Add new environment property	Click “Add Property”; Enter values into fields	“UInt16”; “AGENT_COUNT”; “16384”
3	Add new environment property	Click “Add Property”; Enter values into fields	“Float”; “ENV_MAX”; “25.0”
4	Add new environment property	Click “Add Property”; Enter values into fields	“Float”; “repulse”; “0.05”
5	Open agent dialog box	Hover over “Agent” menu item; Click “Add Agent”	N/A
6	Add a new agent variable	Click “Add Variable”; Enter variables values	“Float”; “x”; “random.random()*25”
7	Add a new agent variable	Click “Add Variable”; Enter variables values	“Float”; “y”; “random.random()*25”
8	Add a new agent variable	Click “Add Variable”; Enter variables values	“Float”; “z”; “random.random()*25”
9	Add a new agent variable	Click “Add Variable”; Enter variables values	“Float”; “drift”; “0”
10	Finish defining agent	Enter values; Click “OK”	Name- “Circle”; Population- “AGENT_COUNT”; Visualise Agent-True; Agent Mesh- “Icosphere”; Mesh Colour- “fcba03”; Mesh Scale- “0.05”

Table F.1: End to End testing of application

No.	Description	Action	Input Data
11	Open message dialog	Hover over “Messages” menu item; Click “View”	N/A
12	Enter message values	Enter values	Name-“location”; Type-“MessageSpacial3D”
13	Add message variable	Click “Add Variable”; Enter values	“ID”; “id”
14	Finish defining message	Enter values; Click “Create Message”	Radius-“RADIUS”; Min-“0, 0, 0”; Max-“25, 25, 25”
15	Add new agent function	Hover over “Agent” menu item; Click “Add Agent Function”; Enter values	“output_message”; “None”; “location”; code in Algorithm 3
16	Add new agent function	Hover over “Agent” menu item; Click “Add Agent Function”; Enter values	“move”; “location”; “None”; code in Algorithm 4
17	Add new host function	Hover over “Agent” menu item; Click “Add Host Function”; Enter values	“Validation”; “Step”; code in Algorithm 5
18	Order simulation execution	Click “Add Layer”; Drag “move” card to layer 2	N/A
19	Configure simulation	Hover over “Run”; Click “Config”; Enter values; Click “OK”	“Circles”; “1000”; “0”; True; “30”; “(30, 30, 30)”; “(0, 0, 0)”
20	Launch simulation	Hover over “Run”; Click “Launch”; Give save location	“CircleSimulation”

Table F.2: End to End testing of application (continued)

## Appendix G

### Test Code

---

**Algorithm 3** “output\_message” agent function code

---

```
FLAMEGPU->message_out.setVariable<flamegpu::id_t>("id", FLAMEGPU->getID());
FLAMEGPU->message_out.setLocation(
    FLAMEGPU->getVariable<float>("x"),
    FLAMEGPU->getVariable<float>("y"),
    FLAMEGPU->getVariable<float>("z"));
return flamegpu::ALIVE;
```

---

---

**Algorithm 4** “move” agent function code

---

```

const flamegpu::id_t ID = FLAMEGPU->getID();
const float REPULSE_FACTOR = FLAMEGPU->environment.getProperty<float>("repulse");
const float RADIUS = FLAMEGPU->message_in.radius();
float fx = 0.0;
float fy = 0.0;
float fz = 0.0;
const float x1 = FLAMEGPU->getVariable<float>("x");
const float y1 = FLAMEGPU->getVariable<float>("y");
const float z1 = FLAMEGPU->getVariable<float>("z");
int count = 0;
for (const auto &message : FLAMEGPU->message_in(x1, y1, z1)) {
    if (message.getVariable<flamegpu::id_t>("id") != ID) {
        const float x2 = message.getVariable<float>("x");
        const float y2 = message.getVariable<float>("y");
        const float z2 = message.getVariable<float>("z");
        float x21 = x2 - x1;
        float y21 = y2 - y1;
        float z21 = z2 - z1;
        const float separation = sqrtf(x21*x21 + y21*y21 + z21*z21);
        if (separation < RADIUS && separation > 0.0f) {
            float k = sinf((separation / RADIUS)*3.141f*-2)*REPULSE_FACTOR;
            // Normalise without recalculating separation
            x21 /= separation;
            y21 /= separation;
            z21 /= separation;
            fx += k * x21;
            fy += k * y21;
            fz += k * z21;
            count++;
        }
    }
}
fx /= count > 0 ? count : 1;
fy /= count > 0 ? count : 1;
fz /= count > 0 ? count : 1;
FLAMEGPU->setVariable<float>("x", x1 + fx);
FLAMEGPU->setVariable<float>("y", y1 + fy);
FLAMEGPU->setVariable<float>("z", z1 + fz);
FLAMEGPU->setVariable<float>("drift", sqrtf(fx*fx + fy*fy + fz*fz));
return flamegpu::ALIVE;

```

---

---

**Algorithm 5** “Validation” host function code

---

```

def __init__(self):
    self.prevTotalDrift = sys.float_info.max
    self.driftDropped = 0
    self.driftIncreased = 0
    super().__init__()

def run(self, FLAMEGPU):
    totalDrift = FLAMEGPU.agent("Circle").sumFloat("drift")
    if totalDrift <= self.prevTotalDrift:
        self.driftDropped += 1
    else:
        self.driftIncreased += 1
    self.prevTotalDrift = totalDrift
    print(f"Drift correct: {100* self.driftDropped /
        (self.driftDropped+self.driftIncreased)}")

```

---