Development of a simulation framework for digital microfluidic biochips

Alexander M. Collignon
Carl A. Jackson
Joel A. V. Madsen

Kongens Lyngby 2022
IMM-B.Sc.-2022

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk IMM-B.Sc.-2022

Abstract

The thesis aims to develop a simulation framework for the Digital Microfluidic Biochip platform (DMF platform) under development at DTU Compute. We present the design and implementation of such a simulation framework, divided into three parts: a simulation engine, logic-based models, and a graphical user interface. The simulation framework is developed to be integrated into the existing web-based programming interface, which utilizes the Blazor framework. The simulation framework is developed as a client-side web application through a combination of WebAssembly, compiled from C#, and JavaScript. Our implementation allows for a non-physical logical estimation of the physical behaviour observed in the DMF platform, designed with modularity in mind, providing an interface for further expansion. We conduct tests on the simulation framework, with the primary evaluation metric being comparisons between the simulation framework and the physical DMF platform. We find that the simulation is very close to what is observed in the physical DMF platform. The framework can therefore be used to conduct experiments, without any cost of failure, before they are executed on the physical platform.

Abstract (Danish)

Målet med denne afhandling er at udvikle et simulations framework til den Digitale Microfluide Biochip platform (DMF platform), som er under udvikling på DTU Compute. Vi præsenterer designet og implementationen af et sådant simulations framework igennem tre dele: en simulationsmaskine, logik baserede modeller og en grafisk bruger grænseflade. Simulations frameworket er udviklet så det kan integreres i en eksisterende web baseret programmerings grænseflade, som gør brug af frameworket Blazor. Simulations frameworket er udviklet som en klientsidet web applikation, gennem en kombination af WebAssembly, kompileret af C#, og JavaScript. Vores implementation muliggør en logisk ikke-fysisk estimation af den fysiske opførsel, set på DMF platformen, designet med modularitet for øje, hvilket gør det muligt at udvide programmet fremadrettet. Vi tester simulations frameworket, hvor den primære evaluations faktor er sammenligninger mellem simulations frameworket og DMF platformen. Vi finder frem til at simulations frameworket opfører sig meget lig den fysiske DMF platform. Derfor kan simulations frameworket bruges til at udføre eksperimenter, uden nogen omkostninger ved fejlede forsøg, inden eksperimenter gennemføres på den fysiske DMF platform.

Preface

This thesis was prepared at the Department of Applied Mathematics and Computer science of the Technical University of Denmark to fulfil the requirements of a BSc in Software Engineering. The thesis was carried out from January the 31st to June the 16th under the supervision of Associate Professor Luca Pezzarossa.

Kongens Lyngby, 16-June-2022

Alexander M. Collignon
Carl A. Jackson
Joel A. V. Madsen

Acknowledgements

We want to acknowledge Luca Pezzarossa, an assistant professor at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark. Luca Pezzarossa is our supervisor for the project, and he has been a tremendous help in reaching the goals of the scope of the project and provided videos and test programs from the physical Digital Microfluidic platform, which we utilized to test the simulation framework.

Contents

Chapter 1

Introduction

This chapter introduces the basic building blocks of our work and explains the aim of our thesis.

1.1 The DMF Platform

The Digital Microfluidics (DMF) platform is an implementation of a technology that allows for the automated performance of simple and advanced biochemical processes in a small and controlled environment. The DMF platform aims to replicate and possibly replace some of the current wet-lab processes by providing a simpler, smaller, and cost-efficient solution. The DMF platform utilizes surface tensions and electrical fields, also called electrowetting, to move tiny droplets of hydrophilic liquid across a surface. The droplets in the platform serve as reaction chambers for possible substances and other organic and non-organic materials, which can then be merged, split, heated, etc. By this, the platform eliminates the need for tubing, pi-petting, and so on, essentially providing all this within itself. In addition, the DMF platform can utilize different components, such as actuators and sensors, to automate and control actions within the platform itself.

1.2 Motivation and Presentation

The DMF platform provides a programmatic execution of biochemical processes; however, an error is bound to exist in such a programmatic approach, being human or computer error. Therefore, it would be favourable to have a testing environment where the programmatic execution can be carried out without risk or resource loss. The obvious testing environment would be software-based, where the execution can be simulated while providing total flexibility, no risk factor, and no resource loss.

This thesis presents and discusses the implementation of such a simulation framework. Implementing a simulation framework for the DMF platform can be split into three key components, a simulator engine, models and a graphical representation of the simulation. The simulation engine should be based on real-world observations and present an essential set of rules which adheres to those observations. The models should contain various calculations to be carried out, and the models should be modular, meaning they can be swapped or edited easily. The graphical presentation of the simulation should represent the actual DMF platform and present all necessary information to the user. At last, all of this should be able to run in the browser in real-time, making it more accessible for the user.

1.3 Specifications

The initial specifications are derived from the project description provided by our supervisor Luca Pezzarosa. This thesis's main purpose, and therefore the main requirement, is to create a simulation framework that can be used as the physical biochip. The simulation framework should support droplets, actuators, and sensors and be modular, adding support for future plug-ins. The simulation framework should tackle three aspects of a simulation, a simulation engine, a graphical user interface, and mathematical models capturing physical and biochemical behaviours. The simulation framework should be integrated into an existing framework, which is a web-based framework, meaning that the simulation framework should both be integrated and also work in a browser.

With the specifications derived from the project description, we conducted a MoSCoW prioritization analysis [1] of the project. The MoSCoW prioritization analysis for the project can be seen in Figure 1.1.

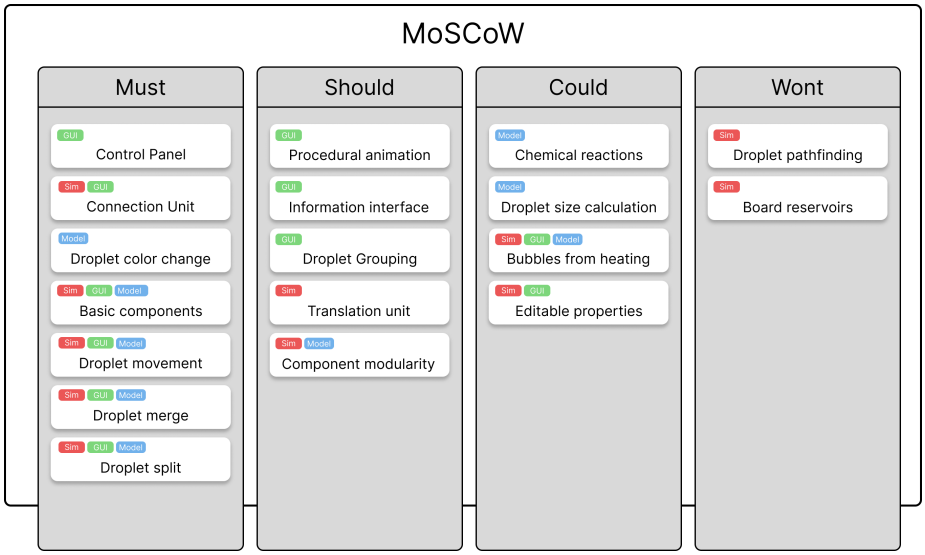

Figure 1.1: MoSCoW prioritization analysis of the simulation framework.

The specifications are respectively labelled with "GUI", "Sim", and "Model" or a combination of them. These indicate which category of the framework they fall into. The specifications in the "should" and "could" tab include basic components and basic movement models for the droplets, if these two tabs are implemented, we should have a functioning basis of the simulation framework with limited features. The MoSCoW prioritization model has been a good method of transforming the scope of the project into a tangible task order. The design and implementation of the specifications is elaborated upon in later sections.

To build the simulation framework, we have utilized the following programming languages.

- C#
- JavaScript
- HTML
- CSS

The web framework, which the simulation framework should be integrated into, is Blazor, which is a web framework that allows the user to create web appli-

cations incorporated with C#. The Blazor version contained in the existing framework is the client-side version, meaning that everything executes solely on the client's machine, a result of using the client-side version is rapid response time, since no data has to be communicated to and from a server instance.

1.4 Thesis Structure

Every chapter in the thesis can be roughly divided into a design and an implementation section. The thesis introduces the project and provides background knowledge about the DMF Platform. The simulation framework describes the interaction between the different components of the program. The models and simulation engine chapters cover the backend design and implementation of the simulation. The graphical user interface chapter covers the frontend design and implementation of the simulation. The Simulation Interchangeability chapter provides a step-by-step guide on how to alter or add components to the simulation engine. The thesis will cover the testing done for the simulation framework and the performance and limitations of the simulation framework. The thesis will discuss problems and ideas that arose during development and finally conclude the final project.

Chapter 2

Background

This chapter describes the background upon which our work is built. It describes the DMF platform and how it works, and it will also present the interface to which the DMF platform adheres. This chapter will also present the frameworks used to build the Simulation Framework and related work in the form of other Simulation Frameworks for the DMF platform.

2.1 The Digital Microfluidics System

The project is based on the Digitial Microfluidic Biochip system developed at the Technical University of Denmark. The system is used to automate the execution of biological and chemical laboratory protocols. This has high cost and performance benefits compared to the traditional wet lab processes. As mentioned in Section 1.1, the DMF platform utilizes surface tensions and electrical fields to move tiny droplets across a surface; this process is called electrowetting on dielectric (EWOD). EWOD works by having a hydrophobic layer underneath a hydrophilic substance. The hydrophilic substance is the droplets that are manipulated. Initially, a droplet will be in the shape of a sphere, but as the electrodes are turned on, the droplet will flatten out because of the electric pull. This can be seen in Figure 2.1.

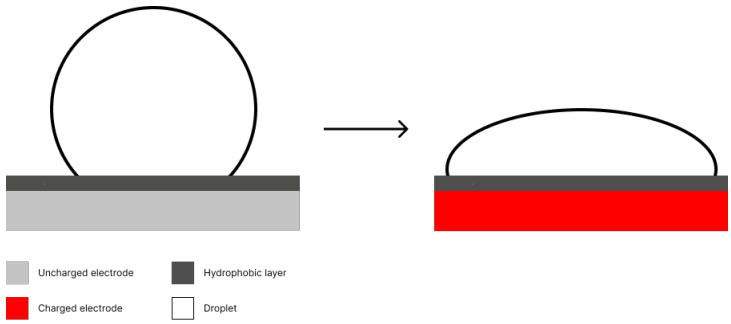

Figure 2.1: The droplet elongates on a charged electrode.

In this project, the focus is on a closed system, meaning that the platform has two layers, one underneath the droplets and one on top of the droplets. The layer on top is an Indium Tin Oxide (ITO) glass, which squeezes the droplets flat, thereby reducing the surface area of the droplet. If the surface area of a droplet is too big, it is unable to split because of the surface tension, even under the influence of a potential voltage. When the ITO glass is put on top, it is considered a closed system where the hydrophobic liquid surrounds the entire droplet in between the layers. This is what we aim to simulate. An example of a split between two insulating layers can be seen in Figure 2.2.

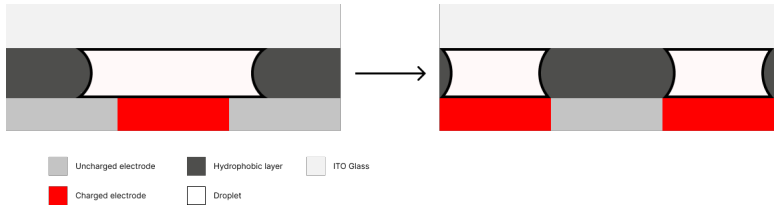

Figure 2.2: Droplet split in a closed system.

With this process, the platform can merge, split and move the droplets to any position on the board.

2.1.1 Platform Components

The DMF platform consists of a board of insulated electrodes in which a potential voltage is acting. Furthermore, the platform consists of components such as actuators, sensors and reservoirs. These are used for heating the droplets,

reading specific values of the droplets and distributing the droplets onto the board. The platform is depicted in Figure 2.3.

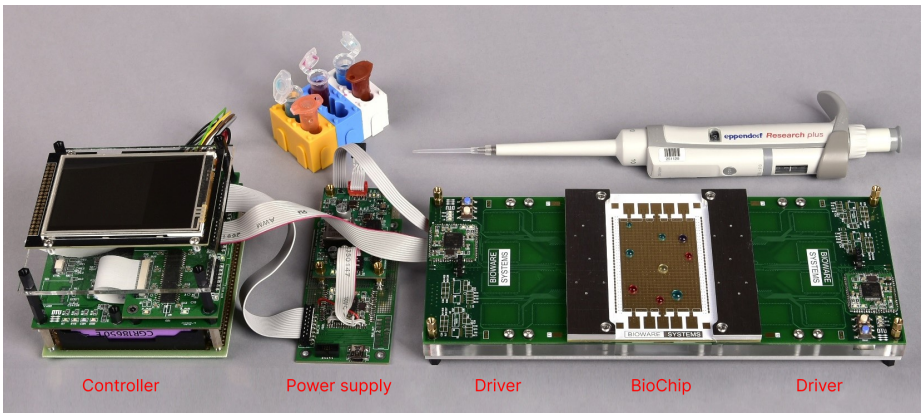

Figure 2.3: The components of the DMF platform are named in the figure.

On the left-hand side of Figure2.3 we see the microcontroller, which executes various programs. The middle component is the power supply for the platform. The right-hand component is the biochip itself, which we in this project strive to simulate. The components in this system are modular and therefore interchangeable. The electrodes are controlled by the drivers on either side of the biochip. Although it is possible to turn on or off one or multiple electrodes at once, the total number of active electrodes is limited. The droplets move to an ON electrode if they are in contact with each other. This is also visible in Figure 2.2.

The components of the platform can be described as the following:

- **information** : contains meta-information about the platform (for example, the platform name, platform type etc.)
- **electrodes** : contains the description of all the electrodes
- **actuators** : contains the description of all the actuators (heaters, pumps, magnets, etc.)
- **sensors** : contains the description of all the sensors (temperature, droplet color, droplet size, etc.)
- **inputs** : contains the description of the platform fluids inputs (reservoirs, tubes, etc.)

- **outputs** : contains the description of the platform fluids outputs (collection reservoirs, tubes, waste pads, etc.)
- **droplets** : contains the description of the active droplets
- **bubbles** : contains the description of gas bubbles (unwanted effect due to evaporation)

2.1.2 DMF Interface

The simulation engine takes input from a virtual machine. The interaction can be seen in Figure 2.4. The lab protocol is translated to the BioAssembly language and then processed by the virtual machine that communicates with the biochip and/or the simulation engine.

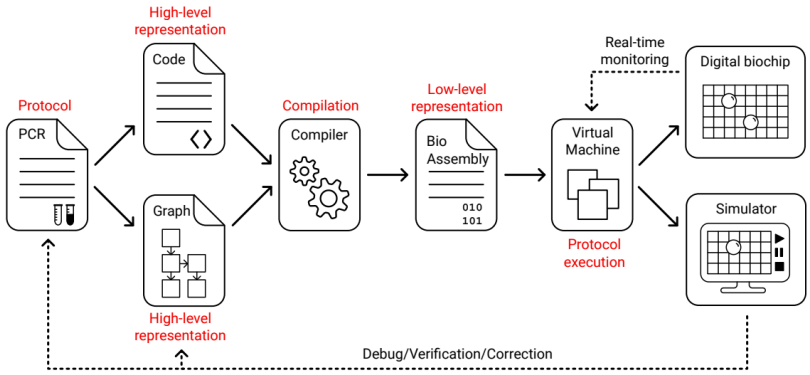

Figure 2.4: From protocol to the simulator. The dotted line represents the output from the simulator and the digital biochip, respectively.

The platform is controlled by an instruction set. This instruction set is described as the BioAssembly instruction set architecture. The instructions include simple actions such as SETEL and CLREL, among others.
The SETEL instruction changes the state of an electrode to on and is used as the command:
`setel <DVR><EL#>[EL#] [EL#] [EL#] [EL#] [EL#] [EL#] [EL#] [EL#] [EL#] \r`
The CLREL instruction changes the state of an electrode to off and is used as the command:

`clrel <DVR><EL#>[EL#] [EL#] [EL#] [EL#] [EL#] [EL#] [EL#] [EL#] [EL#] \r`
where:
<DVR>is an unsigned integer that specifies the driver ID of the electrode you want to set. This is necessary because the electrodes are distributed among two drivers, 0 and 1.
<EL#>is an unsigned integer that specifies the ID of the electrode you want to set.
[EL#] are unsigned integers that specify additional IDs of the electrode you want to set. These are optional, and you can have none or a maximum of 9[2]. These are commands sent to the actual platform.

2.2 Related Works

Although relevant papers about Digital Microfluidic Biochips exist, there are not, to our knowledge, any papers on the full integration of a front- and back-end development of a simulation framework for Digital Microfluidic Biochips. There are, however, papers about the front-end and back-end independently.

Front End for a Digital Microfluidic Biochips Simulator

The thesis "Front End for a Digital Microfluidic Biochips Simulator" [3] introduces the reader to a front-end implementation for a digital microfluidic biochip simulator. Many of the thoughts and design choices are similar to those we considered or made when constructing the graphical user interface for the simulation engine.

A model-based simulation engine for Digital Microfluidic Biochips

The thesis "A model-based simulation engine for Digital Microfluidic Biochips" [4] revolves around the implementation of a model-based simulation engine for Digital Microfluidic Biochips. The project was carried out concurrently with the previously mentioned front-end for a Digital Microfluidic Biochip simulator. The thesis lays out the design and implementation of constructing a simulation engine for the DMF platform in a similar fashion as this project will.

Microfluidic Very Large Scale Integration (VLSI)

This book is authored by Paul Pop, Wajid Hassan Minhass, and Jan Madsen. It introduces the reader to the microfluidics field and microfluidic biochips. It provides an in-depth view of how continuous flow microfluidic biochips work and how these differ from the digital microfluidic biochip mentioned in this book. It also covers applications of biochips in general. [5]

Fault-Tolerant Digital Microfluidic Biochips

This book is authored by Paul Pop, Mirela Alistar Elena Stuart and Jan Madsen. It introduces the reader to the main fluid propulsion principles used by modern microfluidic platforms. In contrast to the aforementioned book, this book provides an in-depth view of how digital microfluidic biochips work and covers several applications of such biochips. [6]

Chapter 3

Simulation Framework

This chapter will present the simulation framework, which is the simulation application structure, determining how the simulation components, simulation engine, models, and graphical user interface, interacts with each other and the user.

3.1 Design

The primary goal of the simulation framework is to provide a simulated reality that correlates as close as possible to what is observed on the physical DMF platform. Therefore, an analysis of the physical DMF platform had to be carried out to achieve this. Analyzing the interface of the DMF platform, seen in Figure 3.4, the run cycle is initialized by a protocol that gets translated into code; the code gets compiled into the bioassembly language, which can then be passed to a virtual machine, that can execute the commands on the biochip. For the simulation framework, every step up to and including the virtual machine is carried out in the same manner as for the actual DMF platform interface, see Figure 3.4. The only difference is that instead of sending the actions from the virtual machine to the biochip, it is sent to the simulation framework. In theory, this interface allows the virtual machine to send the same actions to both the biochip and simulation framework. See Section 8.3 for a comparison between

the biochip and framework. Since we do not have access to the actual virtual machine that is present in the physical DMF platform, we decided to implement a simplified version to showcase virtual machine simulation interactions; we call this virtual machine SimpleVM. By utilizing SimpleVM in correlation with the simulation framework, the application is able to construct a feedback loop, meaning that actions in the simulation engine can cause reactions by SimpleVM, resulting in new droplet routing etc.

To structure the simulation framework, we wanted to utilize a dedicated design pattern and chose to structure the application with the Model-View-Controller (MVC) design pattern.

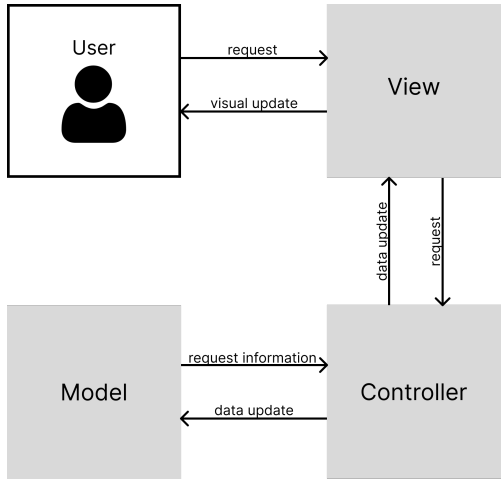

Figure 3.1: Model-View-Controller design pattern with the User.

As seen in Figure 3.1, the MVC design pattern consists of three parts, Model, View, and Controller; the view provides interactivity with the user, and the interactivity action is translated to the controller, which then sends the action to a model, the model updates data, the controller sends the data back to the view, and the data is then reflected on the view. To correlate the MVC design pattern to the simulation framework, the simulation framework can be divided into three components, models, graphical user interface, and simulation engine, directly relating to the MVC (Model-View-Controller) design pattern.

3.2 Structure

The simulation framework is divided into three components, graphical user interface, simulation engine, and models; the communication between the components can be seen in Figure 3.1. The most significant difference between the components is that the programming languages differ; the graphical user interface is written in JavaScript, while the simulation engine and models are written in C#.

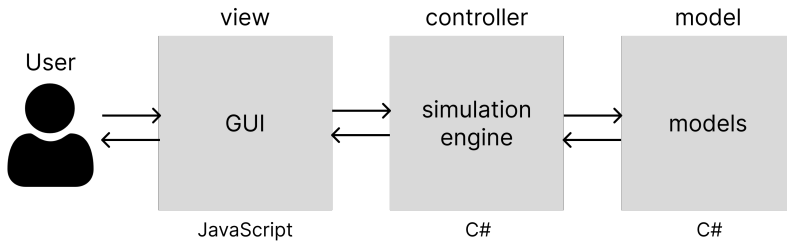

Figure 3.2: Model-View-Controller design pattern with the User.

The difference in programming language presents some issues, which will be discussed in Section 6.2. However, it is necessary to use the p5js framework to visualize the simulation of the DMF platform since this framework is an extension of the JavaScript language.

The simulation engine is responsible for handling everything related to running the simulation, especially keeping track of the simulation data. When a specific action is to be carried out, the simulation engine passes the simulation data to the correct model, which is responsible for modifying the data related to the action. After the data has been modified, the updated data is passed back to the simulation engine, which in return passes it to the graphical user interface, where it is represented as a visual update. This cycle is called an update cycle, it starts with the graphical user interface and ends at the graphical user interface, but at the end state, the data has been updated to reflect the actions. Initially, the update cycle begins by initializing data in the simulation engine but is halted in the graphical user interface, which can then begin the first update cycle. The update cycle allows for the simulation framework to run static programs, but without any feedback loop, as shown in Figure 3.4. To accompany a feedback loop, the SimpleVM has been included in the structure; this can be seen in Figure 3.3.

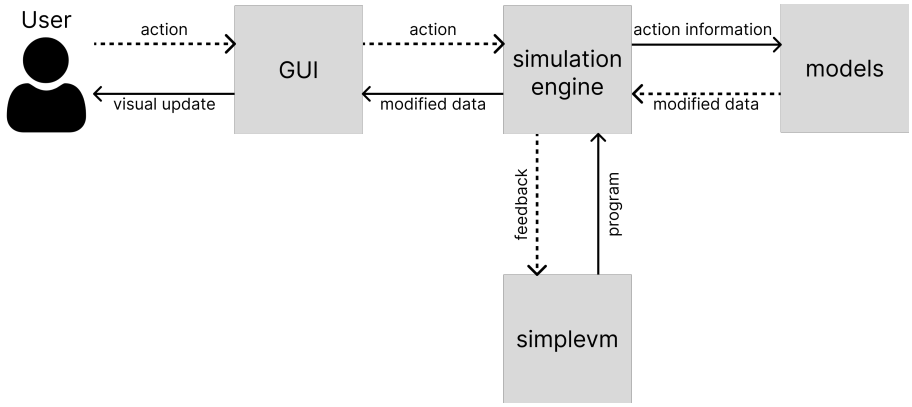

Figure 3.3: Model-View-Controller design pattern with the User.

With the feedback loop, the framework structure allows for reaction to specific observations in the simulation. For example, the observation could be made by a sensor, and the reaction could be a new or additional route applied to the program by the SimpleVM.

This structure is very close to the one observed in the actual DMF platform, and to provide even more reality; we let the components communicate in real-time, meaning that for every action executed by the simulation, a whole update cycle is executed before the following action is loaded.

An abstraction of the communication from Figure 2.4 can be seen in Figure 3.4, where the simulation framework has been modified to be correlative to the one described in this section.

As seen in Figure 3.4, the simulator outputs information about the system, from which the user can construct graphs or correct specific properties if necessary. By utilizing sensors, the virtual machine can monitor components distributed across the virtual board and act upon them.

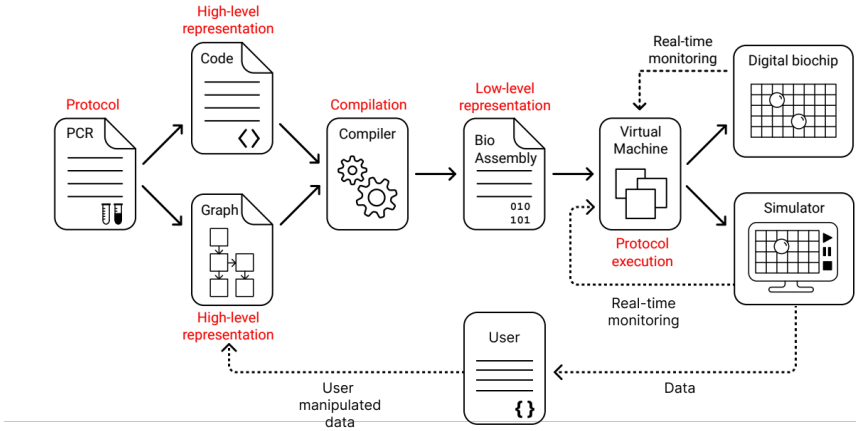

Figure 3.4: From protocol to the simulator. The dotted line represents the output from the simulator and the digital biochip, respectively.

Chapter 4

Models

The models and logical functions are used to run the simulator. This chapter will cover both categories. The models chapter describes all the models that are used to characterize the behaviour of the components in the simulation engine. It is important to note that the models used in the simulation engine do not model the physical behaviours such as surface tension, but that they are made based on observations of the real platform, creating non-physical logical estimation.

4.1 General Model Design

The models should be able to describe the behaviour of the components; therefore, they need to take the container [1] as an input parameter. We decided to use the container as a parameter for all the models because it ensures consistency and because all the models alter the values of the components in the container; therefore, the container can be "updated" in the model itself instead of having to assign container values after running the models. This is also improving the modularity of our simulator; in case a user wants to change existing models,

[1] The container is described in the Appendix entry "Container" A.2.

they do not have to pass in new parameters since all the data is already passed along in the container.

The droplet models should be subscriber-based, see Section 5.3, since they need to be executed based on actions from other components such as actuators and electrodes. On the other hand, the bubble, actuator and sensor models do not need to be subscriber-based since there are much fewer components that they interact with. Therefore, the interaction models should depend on the distance between the two components that interact. This is a simpler design than the subscription design, but it will be just as easy to add new models since the models are executed in the same fashion as the subscription models, see Section 5.1.

4.1.1 Grouping of Droplets

Grouping of droplets is a functionality introduced especially to represent larger droplets of arbitrary size. The droplets used in the simulator are all circular, so the grouping is introduced to represent droplets with non-circular shapes such as an L-shaped droplet or a long row of droplets. Grouping means that droplets will be part of a group of droplets allowing the GUI to draw them as one large droplet when the droplets are connected. An example of L-shaped droplets represented as a group of smaller droplets can be seen in Figure 4.1. All droplets should have a group ID to keep track of what group they are a part of.

As a general rule, all group members must have the same size, but depending on the size of the electrodes the droplets are situated, the size of droplets in a group may vary. They may also have the same colour, temperature, and so on, but all these are decided by the models. The grouping also allows for big droplets, represented as a set of smaller droplets in a group, to have varying temperatures and other attributes among them. For example, if part of a droplet group is on a heater module, this part can heat up quickly and then slowly spread the heat throughout the group.

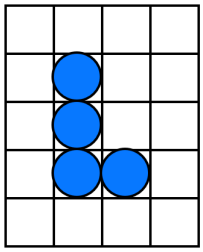

Figure 4.1: An L-shaped droplet is represented as a group of smaller droplets.

.

4.2 Droplet Models

Design

The droplet models contain the logic for altering and moving the droplets, such as colour change, temperature change, evaporation, split- and merge models.

Some of the droplet models should be executed based on a subscription-based execution model, and others should be executed based on their distance to other components in the simulation engine. The design choice of using subscription-based models was chosen since they need to execute on the models depending on the electrodes; since there are approximately 640 electrodes in the current configuration, it would be highly time-consuming to calculate interactions between all the electrodes and every droplet at every execution step. The design choice of using distance-based models for interactions between the droplets and other components was chosen since there are fewer components to calculate between than for the electrodes. Calculating the distance between the droplet and a component such as a bubble or an actuator is the easiest way of deciding which components the droplet needs to interact with at a certain step in the simulation engine.

The droplets should change colour according to other merging droplets; therefore, we want to consider the colour of all the droplets in the group and update the colour of the entire group accordingly. Furthermore, the droplet temperature should be changeable; this should occur when the droplet is on top of a heater element. Since there are only four heaters in the current configuration, we do not have to make this model subscription-based. If the droplet is not on a heater, its temperature should decrease until they reach room temperature. The heating and cooling of a droplet should be based on the content of the droplet. If the droplets are heated to a certain point, they should start evaporating. When

a droplet evaporates, it should produce bubbles, which decreases the size and volume of the droplet.

4.2.1 Design of Droplet Movement

One of the essential elements to simulate on the platform is the movement of droplets. The models that depict these have to be as precise as possible since the core functionality of the platform is the ability to move droplets around.

The first model for moving droplets did everything at once. For example, when an electrode next to a droplet is ON, and the droplet is on an electrode that is OFF, the model will update the droplet's position, moving it to the ON electrode. This model worked fine for moving individual droplets, but when it had to interact with additional droplets, we needed other models to describe when to merge and split. Analyzing the movement of a droplet from one electrode to another, we see that the droplet does not have a gradual or sliding movement from one electrode to another; it instead snaps from one electrode to the other almost instantaneously. We also see that during this short movement, the droplet changes shape. The droplet elongates and covers both electrodes before reshaping into a circular droplet on the new electrode. This movement, comprised of two parts, elongation and reshaping, may be a correct way to model the behaviour of the droplet. The movement can therefore be split into two models. One where the droplet splits from the original electrode to cover both electrodes, and one where the droplet from the original electrode is merged into the droplet on the new electrode. Therefore, the new approach is to split the model of droplet movement into two models: droplet split and droplet merge. This approach will be a more precise modelling of movements, and the new models will, apart from handling the movement of droplets, also handle merge, split, and interaction between droplets.

With the split and merge approach, what happens to the droplets and their groups has to be considered. A split is, with this approach, the act of spawning a new droplet. This means that a newly spawned droplet will have the same group ID as the droplet it comes from. Therefore no new groups are made when a droplet splits onto an electrode next to the droplet. A merge, on the other hand, has the possibility of creating new groups. When a merge happens, a droplet is removed as it is merged into another droplet. The merge can look like a split of a bigger droplet, where the merge divides a group of droplets into two parts. If this is the case, the model must also calculate the new groups created after the merge. An example can be seen in Figure 4.3.

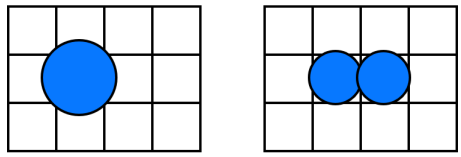

Figure 4.2: A droplet splitting into a neighbouring electrode.

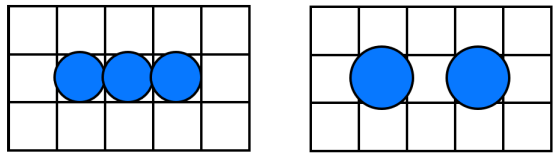

Figure 4.3: In a group of droplets on a line, the middle electrode is turned off triggering a merge of the middle droplet into the two neighbouring droplets, creating two individual droplet groups.

4.2.2 Droplet Split

Design

When designing the model for the splitting of the droplet, we had to take into account the actual behaviour of a droplet on the microfluidic biochip. Observing the droplets' real-life behaviour, we made some observations that the model has to conform to. The observations below are both for moving and splitting a droplet.

- A droplet is not able to be attracted by another electrode if it is not partially overlapping it.
- A large droplet cannot be moved by only one electrode.
- It is possible to split a smaller droplet out of a large droplet.

Addressing the issues one by one, we see that our implementation of the subscriptions helps solve the first listed behaviour. The exact implementation of the subscriptions is discussed in Section 5.3.1. This is because if a droplet is not subscribed to an electrode, it will not be alerted when a neighbouring electrode switches, given that the subscription is based on what electrodes the droplet

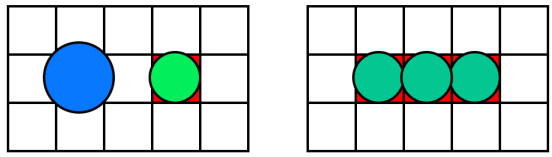

Figure 4.4: A droplet splitting onto an empty electrode joining two groups of droplets together.

overlaps. The implementation discussed in the subscription part which determines if a droplet overlaps an electrode can therefore be used to make sure that a droplet is not attracted by electrodes that it does not overlap. We observed that an electrode is not infinitely strong, meaning it cannot pull a large droplet. Modelling this, we must consider a relation between the number of electrodes pulling on the droplet versus the size of the droplet that is being affected by the pull. Since there is a difference between the size of electrodes, a solution is to look at the total area that the droplet spans and compare it to the total area of the electrodes which are pulling the droplet. A large droplet should also have the possibility to split a small droplet out of itself. This is in the case where a single electrode is not strong enough to move the entire droplet. Splitting a small droplet out of a large droplet on the periphery of the large droplet is possible. For this, we need to determine whether an electrode that we want to split to is on the periphery of the droplet or not. Splitting a droplet can also result in two groups of droplets becoming one. There should therefore be some functionality that checks if there is a change in the groups. This could mean that it seems like droplets are merging, and in reality, they might, but in the abstraction of the program, the droplets will get the same group ID but be two distinct droplets. When two groups of droplets become the same group, some rules also need to be applied; for example, all members of the same group need to have the same colour. An example of the described case can be seen in Figure 4.4.

The action of splitting the droplet out of another droplet, not to be confused with deciding when to split, also needs to follow some rules. As mentioned in Section 4.2.1, splitting does generally not create new groups, but it can join two groups together. It is therefore important to have a functionality to detect if a splitting joins two droplets together. When the new droplet is spawned out of an origin droplet, the droplets after the split must have the same size, according to how droplets in a group have their size calculated. See Section 4.1.1 for a description of grouping.

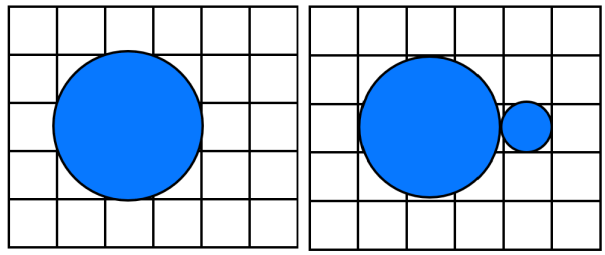

Figure 4.5: A small droplet splitting out from a bigger droplet.

But in some cases, the splitting can create a new group. This is when a small droplet is spawned from a large droplet. If this is the case, the new droplet that is spawned from a larger droplet must be approximately the size of the electrode it is spawned onto. This particular size is chosen since this behaviour is observed on the real board. An example of a smaller droplet splitting out from a big droplet can be seen in Figure 4.5.

Implementation

In one simulation step, the droplet could possibly split onto multiple electrodes. In the implementation, the splitting of the droplet is therefore divided into two parts, first figuring out where to split and then applying the rules outlined in the design section. This is followed by a function that spawns the new droplets keeping data intact and following the rules of subscriptions, grouping and so on.

When looking at what electrodes the droplet has to split to, it is important to remember that a droplet can only split onto an electrode if it overlaps that electrode. This coincides with the implementation of the subscriptions of the droplets; see Section 5.3.1 for the exact implementation. In this implementation of the subscriptions, a droplet is subscribed to all electrodes it overlaps. The droplets hold what electrodes they are subscribed to; this can also be utilized in the split model for finding all electrodes that a droplet overlaps without having to do any calculations.

As discussed in the design part of the split model, there are two types of splitting. The splitting where the group number of the new droplet is the same as the group ID of the origin droplet and the splitting where the droplet that is split out has a new group ID. To distinguish when these cases are applicable, if the electrode we split to is a neighbour of the electrode that the droplet is on, it is a “classical” split. An example of a “classical” split can be seen in Figure 4.2. If

the split is to an electrode that is not a neighbour, it is the case where a small droplet splits out of a bigger one, and therefore it needs to have a new group ID. It is important that if an electrode is completely engulfed by a droplet it makes no sense to split to this electrode, so a check for this is also implemented.

The second part of splitting, once the model has found out where to split, is divided into a couple of steps. First, we will take a look at the "classical" split, where a droplet is split onto a neighbouring electrode. The split itself is divided into a couple of steps. First, when the new droplet is spawned, it inherits most of its origin droplet's data but gets a new ID, a new position, and the volume and size are set to 0. Secondly, the group is calculated. As a general rule, the new droplet will have the same group as its origin, but as described in the design section, the split can connect two groups. The new group is calculated by searching through the electrodes' neighbour list of the electrodes the droplet is located. When the group is calculated, the total volume of all members of the group is distributed equally among all its members while taking into account the size of the electrodes they are located on. Finally, the colour of the droplets in the group is recalculated. The colour is recalculated in case two groups are joined together by the split. If the split is a small droplet splitting from a larger droplet onto an electrode which is not neighbouring the origin electrode, the droplet gets the volume equal to 90% of the electrode it splits onto. The group and colour updates are the same as for the "classical" split.

4.2.3 Droplet Merge

Design

A droplet on the electrode board of the DMF platform will be in the shape of a circle. This is due to real-life surface tension being the lowest for circles. However, droplets can also have other shapes, and this is due to the electric potential of the electrodes. The purpose of the merge model is to figure out when to remove droplets and merge their volume into neighbouring droplets. This is done so the model can describe the behaviour of the droplets trying to remain as circular as possible.

The merge model must determine if a droplet needs to be merged. If the droplet must be merged into multiple other droplets, the model also has to calculate how the volume must be distributed between the different droplets. The volume can be distributed to one or more other droplets. A droplet can merge in two different cases. In the first case, the droplet is on an electrode which is OFF, and it is next to an electrode with a droplet, which is ON. In this case, the

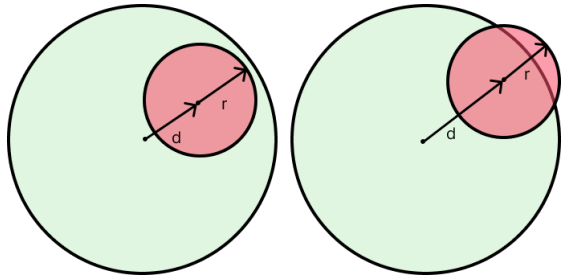

Figure 4.6: Determining if a droplet completely overlaps another droplet, used for merge.

droplet will move onto the ON electrode and merge into the droplet on that electrode. On the DMF platform, droplets can absorb other droplets if they are sufficiently more extensive. If a larger droplet partially overlaps a smaller droplet, the smaller droplet is absorbed if it is on an OFF electrode. If the smaller droplet is on an ON electrode, the droplet must be entirely overlapped by the larger droplet to be absorbed. To determine if another droplet completely overlaps one droplet, we can calculate the distance between the two droplets. If the addition of the distance between them and the radius of the smaller droplet is smaller than the radius of the larger droplet, the tiny droplet is entirely overlapped by the large droplet, see Figure 4.6.

As discussed in the droplet movement section 4.2.1, the merge functionality can change the groups of the droplets on the board. The merge is both able to add and remove groups. Removing groups occurs when a droplet of another group is absorbed. This is often the case when a large droplet absorbs a tiny droplet. However, when neighbouring droplets [2] merge, their group is already the same, as defined by the grouping. The merge can also increase the number of groups. This occurs when a merge causes a group of droplets to be separated so that they must be two individual groups. An example of this can be seen in Figure 4.3. In the figure, a merge happens; this merges the middle droplet into the two droplets on the left and right. When this happens all the droplets are not connected anymore, and they can therefore not be in the same group, so new groups must be made, one for the droplet on the left and one for the droplet on the right.

[2]Neighbouring droplets are droplets situated on electrodes next to each other

Implementation

The merge implementation is comprised of two parts. First, the droplet will check if it needs to be merged into another droplet. This is the case when both droplets are of a similar size and it is the type of merge that occurs when droplets move around and split up. This type of merge is the most common one of the platform. Once this is checked, the droplet will check if any droplets near it are smaller and needs to be absorbed into it.

When merging into neighbouring droplets, the model first checks all the neighbouring electrodes using the neighbour array of the electrode on which the droplet is situated. The model stores how many of these electrodes are ON and how many of the electrodes have droplets situated on them, which are the droplets that will get some of the volume from the droplet which is being merged. If there are one or more electrodes with droplets, which the original droplet must merge into, the original droplet is removed. Cleaning subscribers and other stored data also occur when removing the droplet. The volume is stored and distributed to all the neighbours it merges into. In this case, the droplets are already in a group; therefore, the composition of the droplets is the same. This means that colour and other attributes do not need to be recalculated. Once the droplet is removed, groups for all the neighbouring droplets are recalculated to take care of the case described in the design section and illustrated in Figure 4.3.

The other merge case occurs when a large droplet tries to absorb a smaller droplet. If the droplet is sufficiently big, it uses its subscriptions, which are all the electrodes it overlaps, to check for potential droplets to absorb. If a droplet can be absorbed, according to the rules put forth in the design part, this type of merge also requires the droplet to update its colour according to the droplets it absorbs. This is done by calling the droplet's own colour calculation model, combining the original droplet and absorbed droplets' colour and volume.

Implementation

The colour of a droplet is represented by a hex color code given as a string on the droplet's initialization. When two droplets merge, the colour of the droplets will change; this colour change is implemented in the **dropletColorChange** model. It uses the function **updateGroupColor** to update the colours of the entire droplet group. The function finds all droplets in the given group and calculates their average colour. Afterwards, the function calculates how big a percentile the volume of the merging droplet is compared to the volume of the

group. The colour of the group members is then updated to be the weighted combination of the merging droplet and the group itself. The implementation itself was found on Stackoverflow [7].

To change the temperature of a droplet, we use the model **dropletTemperatureChange**. The droplet temperature model handles the behaviour of the droplet when it changes temperature, either by heating or cooling it, depending on if it is on a warm heater or if it is taken off a warm heater and moved onto the room temperature board. The model checks whether the droplet is on a heater. If the droplet is on a heater, the droplet will be heated according to the heater's temperature. If the droplet is not on a heater, the temperature of the droplet will be decreased until it reaches the temperature of the board, which is currently room temperature at $20°C$. If the droplet is already at room temperature, nothing will happen. The heating and cooling are calculated from the mass, heat capacity, thermal conductivity, area of the droplet and the heat from the heater itself. The parameters mentioned above are based on the substance of the droplet. If a droplet is merging into a group, the temperature change is calculated by the function **updateGroupTemperature**. The function finds all the droplets in the given group and calculates the new temperature of the group. The function considers the percentile of which the group itself and the temperature of the merging droplet contributes to the calculation. The temperature change (ΔT) calculations are based on the mass (m), the temperature (t) and the heat capacity (hc) of the droplets in the group (g) and the merging droplet (d).

$$\Delta T = \frac{((t_{g_1} \cdot m_{g_1} \cdot hc_{g_1}) + (t_{g_2} \cdot m_{g_2} \cdot hc_{g_2}) + (t_{g_n} \cdot m_{g_n} \cdot hc_{g_n})) + (t_d \cdot m_d \cdot hc_d)}{((m_{g_1} \cdot hc_{g_1}) + (m_{g_2} \cdot hc_{g_2}) + (m_{g_n} \cdot hc_{g_n})) + (m_d \cdot hc_d)}$$

When a droplet is heated to a certain point, it will start making bubbles; the model used for this is the **makeBubble**. The model checks whether the following factors are full filled for a given droplet. The droplet is above $90°C$ for more than 0.5 seconds or above $90°C$, and its volume is less than or equal to 10. If this is the case, the model will spawn a new bubble. The model will use 10% of the droplet volume for the split if the droplet is above $90°C$ for 1 second; a droplet with a newly spawned bubble can be seen in Figure 4.7. After creating the bubble, the droplet will decrease in size according to the volume it has given to the newly created bubble. If the droplet's volume is less than or equal to 10, the droplet will also be removed from the simulation, and the bubble will be spawned on the droplet's previous position. Suppose the droplet is above $90°C$ for less than 0.5 seconds. In that case, the model will increment the droplet field "accumulatingBubbleEscapeVolume" [3], which is calculated based on the time

[3]The accumulating bubble escape volume is described in the Appendix entry "Accumulat-

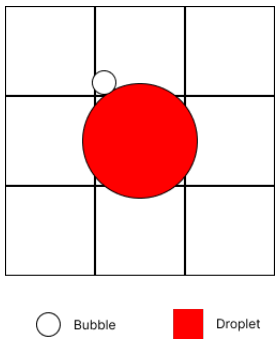

Figure 4.7: A bubble can be seen on the periphery of the droplet.

the droplet has been above $90°C$. If the "accumulatingBubbleEscapeVolume" of a given droplet is bigger than 0.5 seconds, the model will make the droplet spawn a bubble no matter how long the droplet is above $90°C$. The field will be reset to 0.

4.3 Bubble Models

This section describes the models used for bubbles. The bubble models contain the logic needed for adding, moving and merging the bubbles in the simulation.

Design

The bubble models are not subscription-based since the subscription-based implementation is mainly needed for components that interact with many other components to reduce computation time. There will never be too many bubbles because they will merge, and the total number of bubbles will decrease. Therefore the models for the bubbles can be executed at every time step, and they will only have an effect if the bubbles are within a certain distance of the components, such as droplets and other bubbles which they interact with.

The bubbles should be able to merge with other bubbles if they touch each other. Given two bubbles, we decided to merge the smaller bubble into the bigger one because the distance and the size which the bubble has to be updated will change less, and the behaviour of the bubble will be more realistic.

ing bubble escape volume" A.2.

The bubbles should be moved depending on the droplet in the simulation engine. The movement model should be used when a droplet is touching a bubble, hence moving the bubble out of the way of the droplet's trajectory. The bubble movement also has to depend on the droplet group, so the bubble is not moved on top of another droplet in the group. When a bubble is spawned, it should be spawned on the periphery of the droplet in order to mimic real-life behaviour, but it is spawned in the centre of the droplet. Instead of constructing a new model to handle this behaviour, the bubble movement model can be used for moving the bubble to the periphery of the droplet.

Implementation

The model **bubbleMerge** is used to merge two bubbles in the simulation engine. The model takes two parameters, the container and the bubble (caller), which is attempting the merge. The model gets a vector between the caller and every other bubble in the simulation. If the length of the vector is smaller than the addition of the two bubbles' radii and the bubble is smaller than the caller, the bubbles will merge. This means that the caller will absorb the other bubble, increasing the caller's size, and the caller will move towards the direction of the absorbed bubble. If several bubbles full fill the requirements for a merge, the caller will merge with the first found, and the other bubbles will not be considered. If multiple bubbles overlap, the merge will be handled by the merge model in one of the other bubbles. The merging can be seen in Figure 4.8.

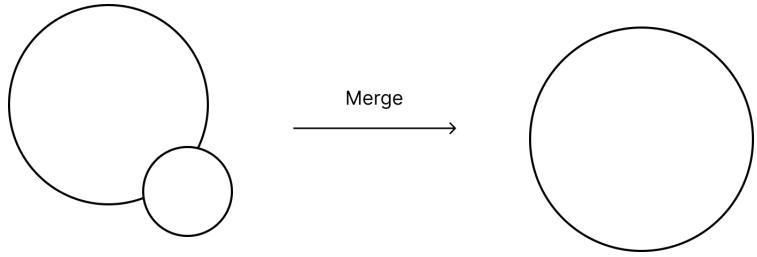

Figure 4.8: A bigger bubble absorbs a smaller bubble, thereby increasing in size and moving toward the smaller bubble.

The model **moveBubble** is used for moving the bubbles in the simulation engine according to the droplets. The model traverses the droplets in the container and checks to see if the bubble is touching a droplet. If the bubble is touching a droplet, it will first move the bubble away from the droplet group and then move the bubble according to the droplet in question.

Moving the bubbles according to a droplet is done with the model **moveBubbleFromDroplet**. The model calculates a vector between the droplet and the bubble. This vector is used to decide how far the bubble should be moved. If the sum of the droplets and the radii of the bubble is smaller than the size of half an electrode, the length of the vector is resized to be the same as the length of 0.6 of the electrode size. This is done so we are sure the bubble is moved away from the periphery of the droplet and away from the electrode on which the droplet is situated. an example can be seen in Figure 4.10. If the sum of the radii is larger than half the size of an electrode, the vector is resized to the sum of the radii. The position of the bubble is then updated by adding the vector to the position of the droplet. An example can be seen in Figure 4.9.

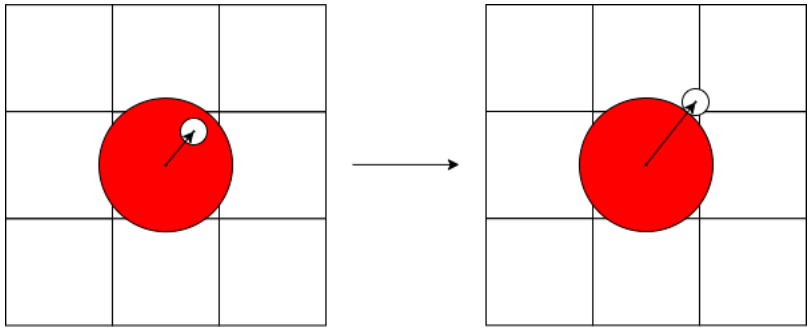

Figure 4.9: A bubble is moved away from the droplet centre by a vector with the length equal to the sum of the radii of the bubble and the droplet.

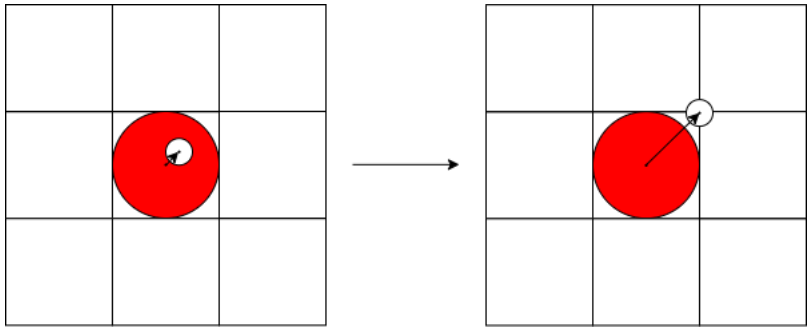

Figure 4.10: A bubble is moved away from the droplet centre by a vector with a length equal to 0.6 times the size of an electrode.

To move the bubbles in the correct direction away from a given droplet, we

call the model **moveBubbleAccordingToGroup** before calling **moveBubbleFromDroplet**. The model **moveBubbleAccordingToGroup** is used so the bubble will not be moved on top of another neighbouring droplet. We could get an infinite loop where the bubble is moved between two droplets in a group if this happens. The model will check whether the droplet has any neighbours given its position. If it has any neighbours, the bubble will be moved a fraction (a pixel) away from the neighbour's position; this will cause the vector calculated in the **moveBubbleFromDroplet** model to point in the "correct" direction, which is pointing away from the group. An example can be seen here:

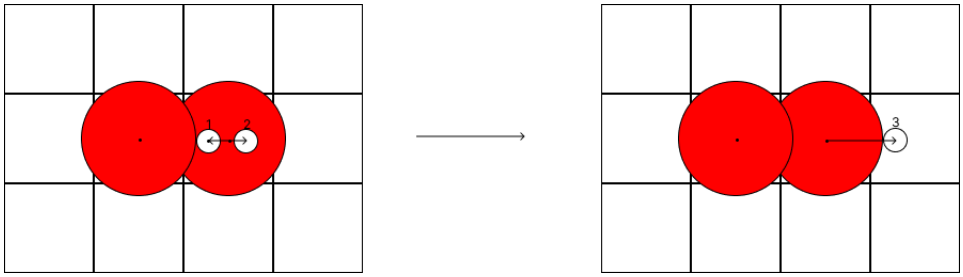

Figure 4.11: 1. The bubble is between two droplets in a group. 2. The bubble has moved a fraction away from the neighbouring droplet. 3. The bubble is moved using the default bubble moving model "moveBubbleFromDroplet".

4.4 Actuator Models

Design

The heater actuator models contain the logic for controlling the heater. The heater should be able to change temperature based on input from the user. The temperature change should be dynamic; therefore, it has to be a "smart heater" which alters its actual temperature based on a target temperature and the power status of the heater. We want to decrease the heater's temperature by using a negative power value since it is easier than calculating the temperature based on the environment around the heater. For further implementations, where the heater, for example, decreases in temperature if a cold droplet is moved on top of it, the design of the temperature decrease should be chosen differently. The default value of the power status is based on the observation that a heater takes 90 seconds to increase from $20°C$ to $90°C$.

Implementation

The model **heaterTemperatureChange** is used to change the heater's temperature. The model sets the power status of the heater. The power status is only changed if there is a disparity between the target temperature and the actual temperature. It only changes after the target temperature has been explicitly set, for the given heater, by, e.g. the virtual machine. The model changes the actual temperature of the given heater using the time and the power status of the heater.

4.5 Sensor Models

Design

The two types of sensors are a colour sensor and a temperature sensor. The sensor models contain the logic used for reading the temperature or the RGB value of a sensor. However, the sensor should only read the value of a droplet if the droplet is situated on the centre electrode of the sensor and there are no other droplets on the periphery of the sensor. This is because a potential virtual machine can push an action queue based on whatever the sensor reads. If the newly pushed action queue attempts to move the droplet away from the sensor, but the sensor keeps reading the droplet, we can end up with an endless loop where infinite similar action queues could get pushed. Therefore we decided on this design choice to avoid this scenario.

Implementation

The temperature sensor can read the temperature of a droplet with the model **temperatureSensor**. The model first checks whether there is a droplet on the sensor, in the sense described in the design section of the sensor models, and if the sensor is a temperature sensor. If both checks hold, the model returns the temperature of the droplet; otherwise, it returns -1,

The colour sensor can read the colour of a droplet with the model **colorSensor**. The model first checks whether there is a droplet on the sensor, in the sense described in the design section of the sensor models, and if the sensor is a colour sensor. If both checks hold, the model returns the RGB value of the droplet's colour in an integer array of length 3; otherwise, it returns [-1, -1, -1].

4.6 Models Summary

In this chapter, we described the design and implementation of the droplet, bubble, actuator, and sensor models, respectively. The simulation engine uses the models to calculate the behaviour of the components at every execution step. After reading this chapter, one should understand the models of each component and how these interact with other components.

Chapter 5

Simulation Engine

This chapter describes the design and implementation of major components in the simulation engine. The simulation engine keeps track of the time and the actions[1] that need to be executed. The actions are stored in a queue in the simulator and whenever an action is executed, the simulator needs to run different models. The models for each component are stored in an arraylist, and the decision of which to run is made by the simulator; this decision is described in Section 5.1. Some components in the simulation engine are dependent on changes in time. Therefore, the simulator needs to handle these changes; this is described in Section 5.2. The droplets' models are run based on a change in electrode status resulting from an action; the dependency is described in Section 5.3. The initialization of the components in the simulation engine is covered in Section 5.4.1, and the communication between a conceptual virtual machine and the simulation engine is described in Section 5.5.1.

[1] An action is described in the Appendix in Section A.3

5.1 Model Chooser

5.1.1 Design

When implementing the simulator engine, the interchangeability of the models was important. The general idea is that the simulator engine should be able to support the future development of more precise models for the simulation. Therefore we made a list of requirements that the simulator should follow when it executes the individual models.

- Models should be able to be changed
- The execution order of the models should be able to be changed and be individual for each droplet
- Droplets should be able to have their unique models

The models for every component in the simulator will be executed at every time step. It will be the simulator's task to choose what order the models are executed. This means that each component holds a list of models, and it is the purpose of the simulator to decide which component should run which model at any given time.

Breadth vs. Depth First

When designing the model chooser, a vital consideration is whether to run the droplets' models depth-first, or breadth-first [8]. This means that if we run all the models for one droplet before running the models for the next droplet, the models can be said to be executing depth-first. On the other hand, if we run a single model for each droplet, for example, split, then run split for the next droplet until all droplets have run the split model, before proceeding with the following model, e.g. the merge model, we are executing breadth-first.

Exploring which to use, we will look at the following example. Two droplets are situated one electrode apart, and both electrodes are off; we then turn on both neighbouring electrodes, the one in between and the one on the far side, where no droplet is residing. The droplets are thus moving back and forth next to each other without merging. See Figure 5.1 for illustrations.

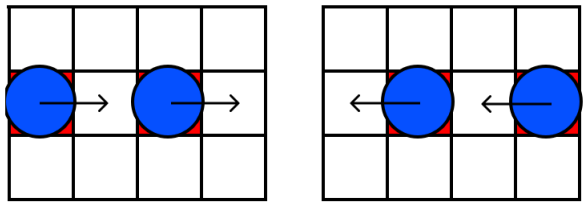

Figure 5.1: Two droplets move back and forth without merging.

We can now explore how the simulation will behave using the two approaches. We first have to choose the order of droplets to execute, beginning with depth-first. Starting with the droplet on the left, it will initially split to the right neighbouring electrode and then merge with the other droplet. When we then want to execute the models for the other droplet, it has been merged with the first, thus not existing anymore. Therefore, the result will be one droplet instead of two droplets moving back and forth. There could also be the case where you start with the droplet on the right. In this case, it would move correctly by first moving the droplet to the right and then the one to the left.

Using a breadth-first approach, the two droplets would first split so we would have four droplets of similar size; then, the merge model would be executed, and the droplets would merge into two droplets in the correct positions. An example can be seen in Figure 5.2, where the final position of the two approaches is shown.

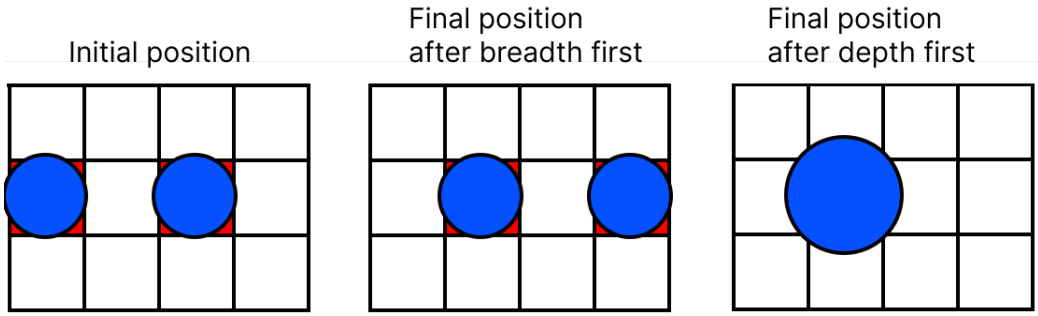

Figure 5.2: The initial position of the droplets on the left and the final positions in the cases using a breadth- or a depth-first approach.

The breadth-first approach gives a more realistic result when analyzing the two approaches. Therefore, this will be the approach we will use when moving forward with implementing the way models are executed in the simulator.

5.1.2 Implementation

For every simulation step, the simulator has a set of subscribers who are either subscribed to the time or the occurring actions. Based on these subscribers, we would like to run the models for each droplet breadth-first while keeping in mind that each droplet can have a different amount and order of its models. This means that when a droplet is created, it can be decided which models it should include and in what order they should be executed. To solve this, we have chosen to organize the subscribers in a queue, and each time a subscriber is reached, they will be allowed to run one of their models. If the subscriber wants to run multiple models, it will have to get in line at the end of the queue.

The approach allows each droplet to run one model, which is chosen by the droplet itself. Each droplet should also be allowed to execute models in its own particular order. To solve this, each droplet holds an array containing the name of a model and an integer that keeps track of which model is the current model. The integer corresponds to an index in the array of model names. The implementation of this can be seen in Code Snippet 5.1.

```
1 public Droplets(
2 ...
3 nextModel = 0;
4 beginOfTimeSensitiveModels = 3;
5 modelOrder = new string[] {"split", "merge", "split", "color", "
      temperature", "makeBubble"};
6 ...
7 }
```

Code Snippet 5.1: Model order array in the Droplets class.

Once a list of subscribers is ready, a loop is used to traverse the subscribers in the queue. The loop runs until there are no more subscribers on the queue. In every loop iteration, the first subscriber in the queue will run one of its models. An example can be seen in Figure 5.3. To choose and run the model, we have made the function **handleSubscriber**. This function takes an integer representing the subscriber and finds the correlating droplet in the container. Once the droplet has been retrieved, the **handleSubscriber** function gets the name of the upcoming model which the droplet needs to run. This model is passed along to a function called **executeModel**. **ExecuteModel** uses a switch statement which switches on the name of the model given; it then calls a function depending on the model that needs to be executed. **HandleSubscriber** will increment the integer **nextModel**, which keeps track of the next model. This means that the next time the droplet has to run a model, it will run the succeeding model in the model order array. If the end of the model array is reached **nextModel** is reset to 0. Implementing this showed that a further requirement was necessary for the

models. The models must return subscribers, which can then be added to the queue of subscribers since additional models might have to be executed. This is how we ensure that the breadth-first approach of running models is achieved. If a model needs other droplets, except for the one having models executed, to be notified of a change, this can be done by adding their ID to the queue of subscribers. This is, for example, done when the splitting model is causing new droplets to be spawned.

5.2 Simulator and Time

5.2.1 Design

The task of the simulator is to keep track of the data, the time, and the actions to be executed. It is also the task of the simulator to tell which droplets, electrodes and so on when to run their models.

The simulator takes actions as the primary input. An action is, for example, turning on or off an electrode. It is then the simulator's goal to simulate the droplets' position with the help of the models. All actions hold a time at which they are executed. The simulator should also be able to keep track of the current time and execute the actions at the correct time; this means that if multiple actions hold the same time, they are executed simultaneously.

The original idea was that the simulator would only update upon actions. However, we found this is not always sufficient since a platform user might use a sensor to wait for a droplet to reach a specific target temperature during the implementation. Thus, we explored the idea of updating the simulator at every particular time step. This particular time step could be every second or millisecond; in the future, this time step will be referred to as delta time.

Because the Graphical User Interface (GUI) is handling the program's front-end, the simulator should be able to be put into action on a request from the GUI. The GUI should decide how long it wants the simulator to run. Therefore the simulator should take input on demand from the GUI, which informs the simulator about how long it needs to run. This gives rise to a potential issue in the case where a droplet is situated on a heater, and both the heater and the droplet are $20°C$. The heater is then heated to $80°C$. The GUI then asks the simulator to run for 100 seconds, but if the simulator only executes once within 100 seconds, two things will happen. If the heater's model is executed first, the heater will have reached the desired temperature of $80°C$. Then the droplet will

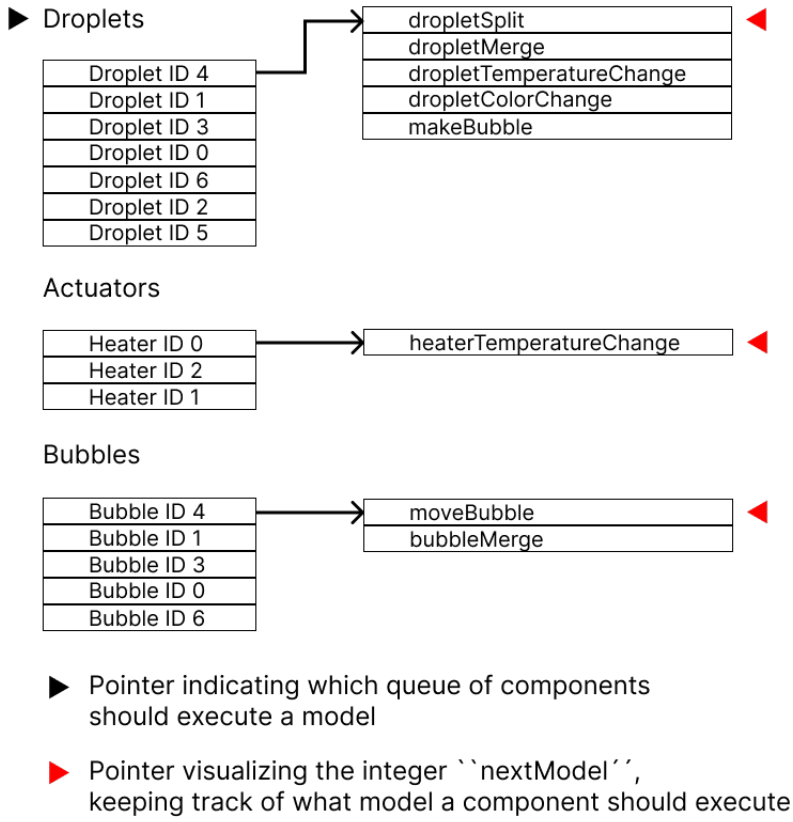

Figure 5.3: The figure illustrates the data structure of the simulator engine in regards to the queue of components and how a model is chosen. At each iteration, the simulator takes the first item in the component queue, finds the component's model array and executes the model which the pointer "nextModel" is pointing to.

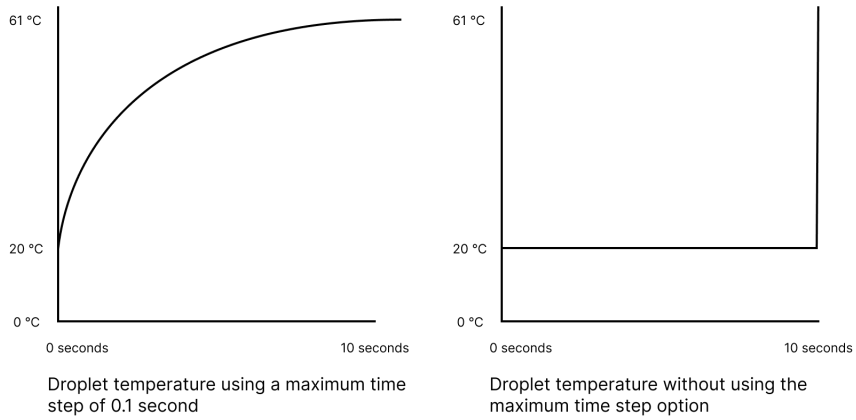

Figure 5.4: Droplet temperature change in a 10 second interval. On the left hand side, the maximum time step is set to 0.1. On the right hand side the maximum time step is not set.

be executed, and it will be as if the droplet has been on the $80°C$ heater for 100 seconds, resulting in wrong behaviour. In reality, the droplet should have only been exposed to the heating for a few seconds. In the case where the droplet is executed first, the resulting temperature of the droplet is going to end up at $20°C$ because the model of the heater is executed after the droplet has moved around.

To solve the issue of models being dependent on each other and the time, we set a maximum time step in the simulator. This means that the simulator has to execute at least every specific time step. The user can still simulate any time interval they want, but if the user wants to step 100 seconds, the simulator will split the 100 seconds into several smaller time steps. This small time step is decided by the maximum step set in the simulator. For example, suppose the user requests 10 seconds of execution and the maximum time step amount is 0.1 seconds. In that case, the simulator will execute 100 times within those 10 seconds before returning the result to the GUI. An example of the temperature of a droplet situated on a heater of $90°C$ can be seen in Figure 5.4.

Apart from allowing the user and the GUI to run the simulator until a specific time, it was beneficial to make it possible to run until the following action of the action queue occurs. Thereby, the user does not have to know which time a specific action happens, but they can tell the simulator to run until the following action has been executed.

Given that the simulator can run at any given time step, this time must relate to

the time of the actions in the action queue of the simulator. If given an example where the GUI asks for a time increment of 10 seconds and there is an action with a time stamp of 5, the simulator has to execute the action at timestamp 5 and then proceed with the remaining 5 seconds.

5.2.2 Implementation

The simulator has a constructor initialized by the GUI at the start of the simulation. The constructor creates an instance of the simulator and uses the initialize function, which is covered in Section 5.4.1, to initialize the subscriptions, data, etc.

The time of the simulator is stored in the container as a float named **currentTime** and a float named **time step** where **currentTime** is the time the simulator currently at and **time step** is the delta time. The reason for storing the times in the container instead of the simulator and having two different representations of time is that the models that depend on time can access the current time and the delta time.

In the following part, the working of the simulator engine will be discussed; these are illustrated in Figure 5.5. The **simulatorStep** function is the function used by the GUI to call the simulator and ask for the simulation to run until a certain time. When **simulatorStep** is called, the function sets a variable target time to be the addition of **currentTime** and the time step of the simulator. The simulator then instantiates the while loop in Figure 5.5 that runs until the target time is reached.

Every time the while loop in Figure 5.5 runs, it checks if there is an action at **currentTime**. If this is the case, the simulator has reached one or more actions that must be executed using the approach discussed in Section 5.1. When executing the actions, a complete list of subscribers is returned. If it is not the case that **currentTime** correlates to an action on the queue, the subscribers will be reset to the subscriber list in the container.

Once a list of subscribers has been retrieved and stored in a queue, the execution of the models proceeds. The execution of the models is elaborated in Section 5.1. The simulator then takes the time increment into account, and this increment is dependent on two cases. If the subscriptions returned are from the execution of an action, the time is 0. If the subscriptions come from the subscription list in the container, the time must be greater than 0.

To find the exact time, one must check if there is an action between the current

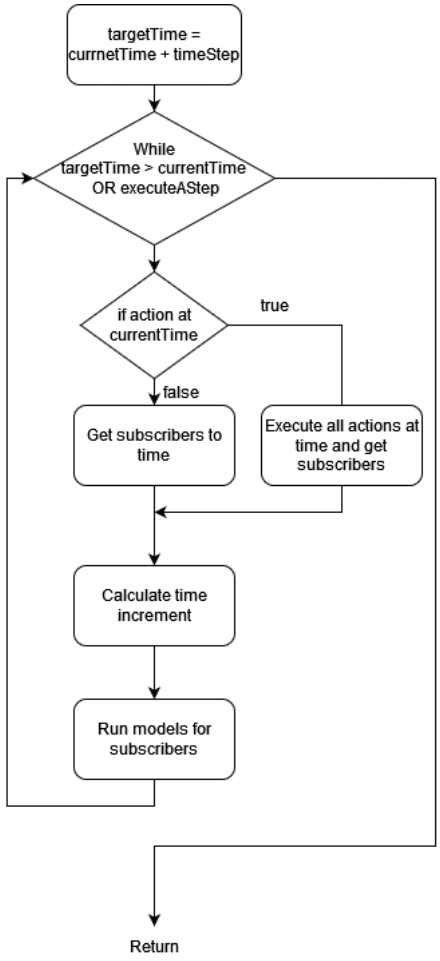

Figure 5.5: Simulator engine while loop.

time and the target time. If there is an action, the simulator will run, but the time of the simulator will, at maximum, be run until the time of the action. When this happens, a boolean **executeAStep** is set to true. This allows the loop to continue even if **currentTime** is equal to the target time, since an action with a time equal to **currentTime** must be in the queue. If there is no action, the maximum allowed time is the target time. Once the time step is calculated, it is split up into smaller time steps to ensure a precise simulation, which is especially necessary for large time increments.

5.3 Subscriptions

Design

To improve the computation time of the simulator, we have chosen to use an event-based implementation. An event-based implementation means that components such as droplets will only be called when necessary. This means that an action on one side of the board will not necessarily require droplets on the opposite side of the board to be recalculated. Still, by changing an electrode next to a droplet, we want the droplet models to be executed since this would cause the droplet to move. The general idea is to limit the computation to the absolute minimum and only calculate where we expect changes. To implement this, all components, such as droplets, bubbles, etc., will be able to subscribe to specific actions or other components and be alerted when necessary. First, we must define what a particular component should or can be subscribed to. Droplets should be able to subscribe to electrodes; this means that if an electrode is powered on or off, it will notify the nearby droplets. All the components in the container will also be able to subscribe to the simulator's time since some models might be time-sensitive and have to be executed often, for example, models describing the temperature.

Implementation

Implementing the subscriptions simultaneously with the simulator engine and the models led to a lot of changes for the subscriptions as new features and functionalities were added to the simulator engine.

The initial implementation of the subscriptions was added when the only features were the droplets on the board and their initial movement models. In this

case, electrodes held an arraylist with the id of the subscribed droplets. When a droplet is added or moved, it calls a model to initialize subscriptions; see more in Section5.3.1. This model subscribes the droplets to all the electrodes it has to be notified by. It also clears its old subscriptions. In the original subscription models, this was purely based on the neighbours of an electrode; therefore, the droplet would be subscribed to the electrode it was on and the electrodes given neighbours. During the implementation, we figured that this was not an optimal solution since a droplet could be larger or smaller, thereby overlapping more or fewer electrodes than initially assumed. The exact functionality of this model is explained in Section 5.3.1.

With the development of the simulator and the addition of bubbles, actuators, and delta time, the way the subscriptions worked had to be revised. The addition of delta time allowed the simulator to step an amount of time and not necessarily to the time of the following action. This meant that the droplets that were sensitive to changes in time had to be notified. An example of this could be the changing of temperature. This proved to be a significant change to the way the subscriptions worked. The original idea with the subscriptions was to reduce the computation needed, but if all droplets had to have all their models run at every time step, it would ruin the purpose. To preserve the subscriptions' computational advantage, we had to reconsider the types of models that the droplet uses. The following section will distinguish between models subscribed to actions, meaning models that execute based on subscriptions to an electrode and models executed based on a change in time. This allows for fewer models to be run. The exact implementation will be elaborated upon in Section5.1.

5.3.1 Subscription Model Droplets

To subscribe the droplets to an electrode, we made a model so that the user, in the future, will be able to change the requirements of a subscription.

The first implementation of the subscriptions for droplets was based solely on the neighbours generated for the electrodes. In this implementation, a given droplet would notify the electrode it was on top of, along with the neighbouring electrodes, that it wanted to be notified if a change happened. In addition, the droplet would also hold the index of the electrodes that it was subscribed to in order to notify them again once the droplet moved, thereby clearing the subscriptions.

This implementation worked as long as all the droplets were of relatively small size, covering a few or one electrode. When implementing models for splitting and merging droplets of a larger size, this implementation of subscriptions

proved insufficient. It was insufficient since a droplet covering multiple electrodes would only be subscribed to the electrodes near the droplet centre. The subscriptions of the droplets need to be based on the electrodes with which the droplet overlaps; this means that a large droplet will subscribe to more electrodes than a small droplet. To implement this new version of the subscriptions, the function **DropletOverlapElectrode** was made. The function determines whether a droplet overlaps an electrode using geometric calculations. This is necessary since a given droplet only knows its centre and radius, and a given electrode only knows its position and size. Without this function, it would be necessary to check every pixel[2], which is extremely slow.

Therefore, the final implementation includes an algorithm that determines whether a circle overlaps a rectangle or not. The first approach was to look at the corners of the rectangle and the centre of the circle and then check whether the distance between these was smaller than the droplet's radius. If this is the case, it can be determined that the circle is overlapping the rectangle. An example can be seen in Figure 5.8.

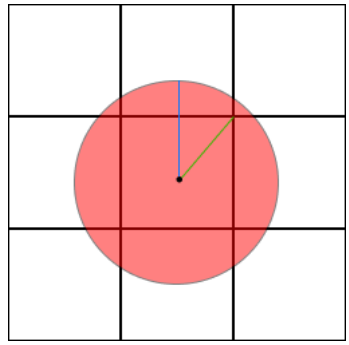

Figure 5.6: In this example, the droplet fully overlaps the corner of the neighbouring electrode.

This is an error-prone implementation. An example where the droplet overlaps parts of the rectangular electrode, but none of the corners would not be determined as an overlapping droplet. This is visualized in Figure 5.7.

[2]A pixel is described in the Appendix entry "Pixel" A.2

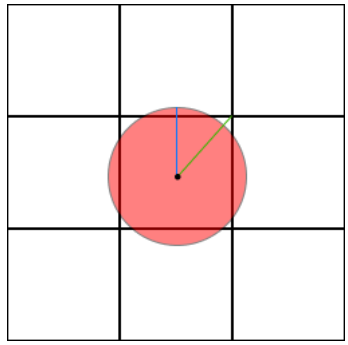

Figure 5.7: In this example, the droplet overlaps the electrode, but it does not overlap the corners of the electrode and is therefore determined as not overlapping the electrode.

The new approach is to get the distance from the centre of the droplet to a line segment. By this, it can be determined whether the droplet overlaps the electrode correctly. The idea is, therefore, to look at all the corners of the rectangle, determine the closest two, and find the distance from the centre of the droplet to the line segment created by the two nearest points. An example of the new algorithm can be seen in Figure 5.8. The implementation of the distance from a point to line segment was inspired by an existing article [9].

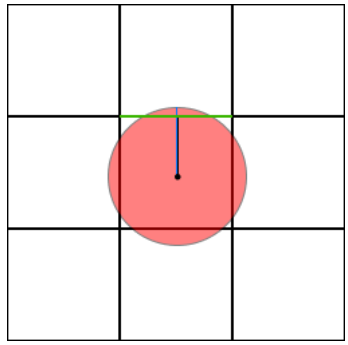

Figure 5.8: In this example, the droplet overlaps the electrode, but it does not overlap the corners of the electrode. This is because the distance from the droplet centre to the line segment between the corners of the electrode is smaller than the droplet radius. The droplet is therefore determined as overlapping the electrode.

This implementation also works for electrodes of a shape that is not rectangular,

but with one limitation, the two chosen points must be next to each other. With the new implementation, it can easily be determined whether a droplet overlaps an electrode or not. An example of a droplet which does not overlap an electrode can be seen in Figure 5.9.

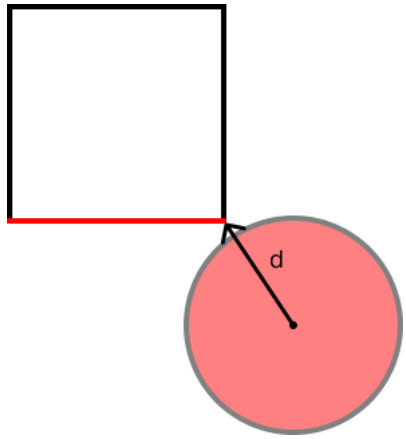

Figure 5.9: "d" is the distance. Since the corner of the electrode is outside the periphery of the the droplet, the droplet is not considered to be overlapping the electrode.

To generate the subscriptions, we make use of the previously mentioned function and the neighbours of the electrodes, which are found when initializing the simulator 5.4.1. The initialization of the subscriptions runs recursively, starting with the electrode the droplet is on. It uses the neighbours of that electrode and checks whether the droplet also overlaps them. If it overlaps the neighbour, the function does recursion again, starting from the neighbour's neighbour and so on. If the droplet does not overlap an electrode, the recursion will end.

5.4 Initialization and Setup

5.4.1 Initialize

The data is initially loaded into the container datatype given a JSON file in the simulator, but many of the extra functionalities have not been initialized. Therefore there is a need for multiple functions to initialize the data. The

initialize class contains all the logic needed to initialize the simulator given data [3] from the initial container.

The function **initialize** takes two parameters; an array of electrodes with their neighbours [4] (if such has been provided in a JSON file) and the container. A few of the functions used by the initialize function are sorting between the type of component used in the function. For example, if a sensor is initialized, the function determines whether it should initialize a sensor of the RGB type or the temperature type.

The functions that add extra fields or variables to the elements in the container are listed below.

initializeBoard takes two parameters, an array of electrodes with neighbours and the container. The function adds the neighbours to the electrodes in the container and returns the new electrode array with their neighbours.

initializeDroplets is assigning group numbers to all the initial droplets on the board; each droplet is assigned a group number according to its index in the droplet list from the JSON file.

initializeActuators is assigning specific fields based on the type of actuator; there is currently only one type of actuators, a heater. A heater extends the actuator class and the fields *valueActualTemperature, valueDesiredTemperature* and *valuePowerStatus*, therefore need to be initialized.

initializeSensors is assigning specific fields based on the type of sensor; there are currently two types of sensors, an RGB sensor and a temperature sensor. The two sensors extend the sensor class. The extra fields that need to be initialized in the RGB sensor are *valueRed, valueGreen* and *valueBlue*. The extra field that needs to be initialized in the temperature sensor is *valueTemperature*.

After running these functions, the droplet, bubble and actuator IDs will be added to the container fields “subscribedDroplets”, “subscribedBubbles”, and “subscribedActuators” respectively. These fields are used for handling the models of these components and the event-driven parts of the simulator. If the given array “electrodesWithNeighbours” is null, the initialize function will use the function “findNeighbours” to initialize the neighbours of every electrode. The run time of this function is relatively slow; therefore the initialization will take significantly longer if the array “electrodesWithNeighbours” is not initially given. Before returning the container, which will contain all initial DMF plat-

[3] The data is described in the Appendix in Section A.3.
[4] A neighbour is described in the Appendix entry “Neighbour” A.2.

form data, the initialize function will call **initializeSubscriptions** which handles the subscriptions for all the droplets. The function traverses the droplets and uses the function "dropletSubscriptions" to initialize the subscriptions for each droplet.

5.4.2 Neighbourfinder

This class [5] contains the functions used for finding the neighbours of the electrodes. The initial JSON file passed to the simulator might not have any information about the neighbours of the electrodes. It is possible to change to JSON file to include neighbours, resulting in the neighbourfinder not being run, saving approximately 18 seconds for the initialization.

The first time a new configuration is introduced in a JSON file, the neighbourfinder will be called when initializing the simulation engine. The simulator will then write a JSON file with electrodes with neighbours to the console. The user can then copy, add to the repository and change the "electrodesWithNeighbours" to be that of the newly created neighbour file.

The neighbourfinder works by traversing the electrodes in the container and matching on their shape [6]. If the shape of an electrode is a rectangle, the logic for finding neighbours of rectangular electrodes is used. If the shape of an electrode is an arbitrary polygon, the logic for finding neighbours of polygons is used. Both functions return an arraylist of electrodes, which are the neighbours of the given electrode.

We use two helper functions for finding the neighbours of the electrodes. The first one, **isCorner**, takes a point and an electrode. The function is only used for polygons; it traverses the corners of the polygon and checks whether the given point is a corner in the polygon or not. The second helper function, **gcd** takes two integers as input and returns the greatest common divisor of the integers. [11]

Finding the neighbours of rectangular electrodes proved easier than finding the neighbours of polygon-shaped electrodes. The function *findNeighboursByElectrode* traverses the electrodes and tries to find a matching neighbour of the electrode we are looking at. The function does this by defining a point in the top left and bottom right corner of the electrode. It then traverses each side of the electrode by incrementing or decrementing the x or y value of the position as shown in Figure 5.10.

[5] A class refers to the C# class [10].
[6] The shape is described in the Appendix entry "Shape" A.2.

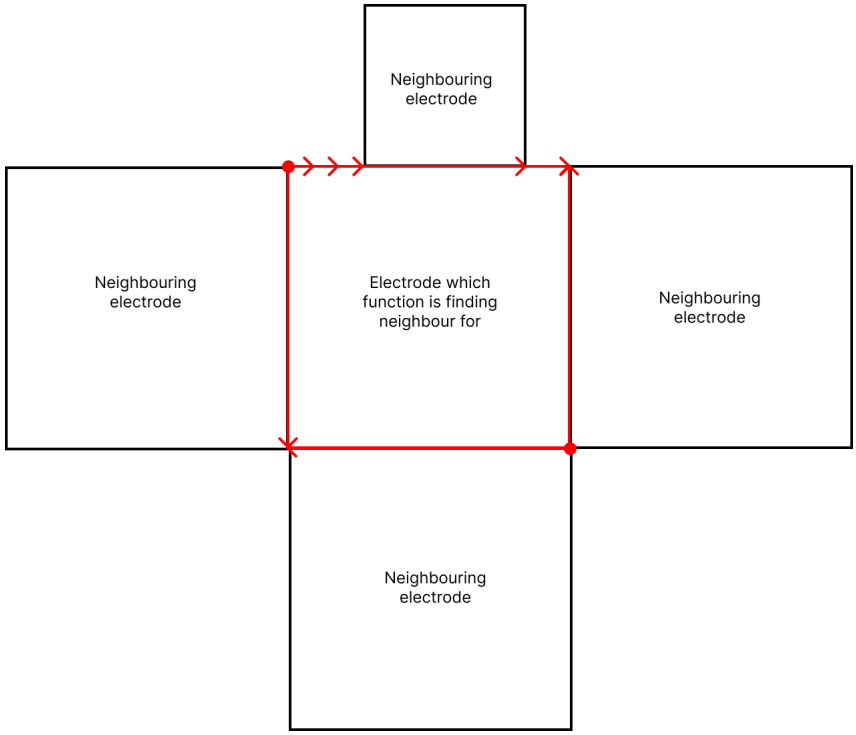

Figure 5.10: Traversing an electrode and finding neighbours.

If a neighbouring electrode is found at the point the function is currently looking at; the search point will be incremented or decremented by the size of the found electrode, thereby traversing the electrode faster. If no neighbouring electrode is located, the side is traversed by 1 pixel at a time. All the neighbours are added to an arraylist stored for the given electrode.

Finding the neighbours of polygon-shaped electrodes is more time-consuming, as the sides that need to be traversed can be at any given angle. For a given electrode, the function will calculate unit vectors from every corner to the next corner in the corner array. The polygon sides will then be traversed from corner to corner until every side has been looked at. If an electrode is found at the point the function is searching; the electrode will be added to the neighbour array. If the electrode is rectangular, the given polygon electrode will also add itself to the neighbour array of the rectangular electrode. This means the rectangular electrodes do not have to search for polygon-shaped neighbours, and all the polygon electrodes will still be added to the rectangular electrodes' neighbour arrays. An example of this can be seen in Figure 5.11.

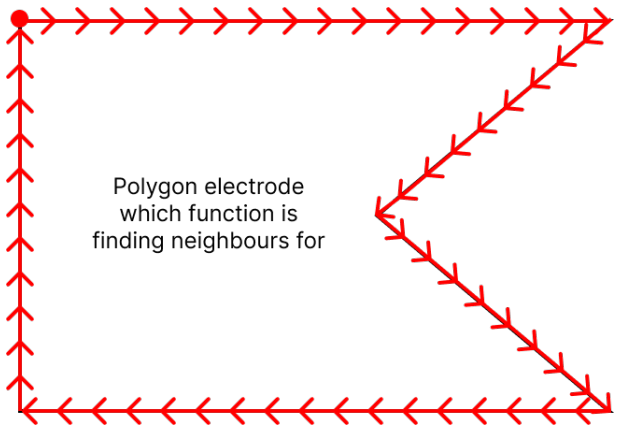

Figure 5.11: Traversing a polygon-shaped electrode using unit vectors.

5.5 SimpleVM

5.5.1 Design

To demonstrate how a third-party software, such as a virtual machine, can interact with the simulator, we developed a so-called SimpleVM (short for a simple virtual machine). We did not implement a "smart" Virtual Machine since this was not a part of our specifications and not within this project's scope. The SimpleVM should be able to do some simple interactions with the simulator at runtime. The design is relatively straightforward since this is merely a demonstration of how one could implement a virtual machine. The SimpleVM should be able to send actions to the Simulator Engine given information about the components in the simulator. This can be seen in Figure 5.5.1.

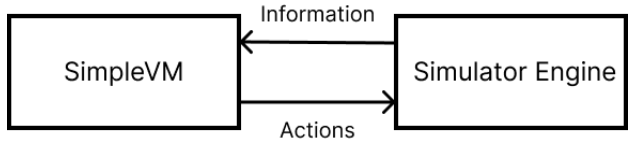

Figure 5.12: Example of VM interaction.

The SimpleVM should be able to read from components in the Simulation Engine, such as sensors. Furthermore, it should be able to control actuators, e.g.

turning them on, by providing a target temperature. Finally, it should also be able to change the action queue of the simulator. To achieve the functionality of the SimpleVM, we can define a list of functions for the SimpleVM.

```
/* SimpleVM fields */
// References the Simulation instance
type simulator                  = Simulator
type actionQueue                = ActionQueueItem queue
type time                       = float
type desiredTemperature         = float
type heaterID                   = int
type sensorID                   = int
type RGB                        = (int * int * int)

/* SimpleVM functions */

// Actuator
function turnOnHeaterAtTime: time * desiredTemperature * heaterID ->
                                void


// Sensors:
function readRGBValueOfColorSensorAtTime: time * sensorID -> RGB
function readTemperatureOfTemperatureSensorAtTime: time * sensorID ->
                                float

// Interactions
pushActionsAtTime: time * queueToPush -> void
```

Code Snippet 5.2: SimpleVM data and functions, listed in a descriptive pseudo code.

The functions are meant to have the following functionalities.

- turnOnHeaterAtTime - This function is used to turn on a given heater at a specific time point in the simulation, increasing the heater's temperature.
- readRGBValueOfColorSensorAtTime - This function is used to read the colour value of a droplet through an RGB sensor at a given time point in the simulation.
- readTemperatureOfTemperatureSensorAtTime - This function is used to read the actual temperature of a temperature sensor at the given time.
- pushActionsAtTime - This function adds new actions to the running action queue in the simulation, resulting in an updated simulator action queue.

5.5.2 Implementation

The interaction with the simulator works by calling the function **doApiCall** at every simulator step. The function allows the SimpleVM to alter the temperature of the actuators, read the values of either an RGB or temperature sensor, and it pushes user-generated action queues to the running action queue of the simulator based on the values read.

The SimpleVM can set the target temperature of a heater at a certain time with the function **turnOnHeaterAtTime**, which takes a time stamp, the desired temperature, an ID for the heater and a boolean. The function checks whether the simulator is at the correct time and whether heaterCalled is false before calling the simulator function "setActuatorTargetTemperature". The correct time is when the time given to the function is less than or equal to the current simulation time. If it is less than the simulation time, we have passed the time step of which the heater should be called. To handle this, we use the boolean heaterCalled, so the target temperature is only set once. **setActuatorTargetTemperature** takes a heater ID and a float representing the desired value of the target temperature. If a heater with the ID exists, its field "valueDesiredTemperature" will be changed to the desired value.

The SimpleVM can read the RGB value of a color sensor at a certain time with the function **readRGBValueOfColorSensorAtTime**, which takes a timestamp and an ID for the color sensor. The function checks whether the simulator is at the correct time and whether rgbSensorCalled is false, as we did in the turnOnHeaterAtTime function, before calling the simulator function "getColorOfSensorWithId". **getColorOfSensorWithId** takes the ID for a sensor and tries to find the sensor in the container. If a colour sensor is found, the colour sensor model "colorSensor" is called and the RGB value of the colour sensor is assigned based on the returnvalue (an array of integers representing the RGB value of the sensor) of the "colorSensor" model. In the end, function returns the RGB value of the colour sensor to the SimpleVM.

The SimpleVM can read the temperature value of a temperature sensor at a certain time with the function **readTemperatureOfTemperatureSensorAtTime**, which takes a time stamp and an ID for the temperature sensor. The function checks whether the simulator is at the correct time step and then calls the simulator function "getTemperatureOfSensorWithID". **getTemperatureOfSensorWithID** takes the ID for a sensor and tries to find the sensor in the simulator data container. If a temperature sensor is found, the temperature sensor model "temperatureSensor" is called. The function checks whether there is a droplet on the sensor with the function "getDropletOnSensor"; if this is the case, the sensor will read the temperature of the droplet; if not, the read value

will be -1.

With a simple check on the return value of the sensors, the SimpleVM can add actions to the original action queue at runtime. To demonstrate the SimpleVM, we have currently created four distinct action queues. By using the described functionalities of the SimpleVM, we can now set up an example with the red action queue, which is meant to be pushed when the RGB sensor senses a red droplet. The SimpleVM is reading the RGB sensor, when the color read by the RGB sensor is equal to the RGB value of red (255, 66, 0), the action queue "redQueuePush" will be inserted into the original action queue; an example of the sensor reading the red droplet can be seen in Figure 5.14. This is done with the function "pushActionsAtTime"; an example can be seen in Figure 5.13, which takes a time stamp and an action queue where it adjusts the time of execution for the actions to match the time stamp. In case the SimpleVM is intending on pushing the same action queue multiple times within the same run, the function will create a deep copy[7] of the action queue and adjust all the actions time step to match the current time step of the simulator. The function then calls the action queue model "pushActionQueueToOriginalActionQueue", which intertwines the actions in the queue with the actions in the original queue currently running in the simulation.

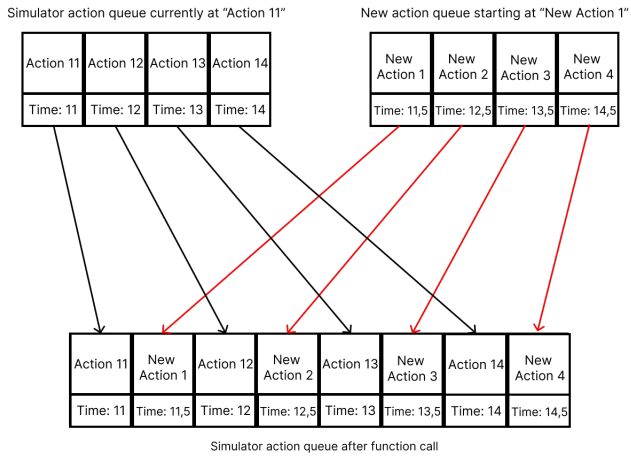

Figure 5.13: Example of function call "pushActionsAtTime(10, new action queue)".

Figure 5.14 represent the board before and after a newly pushed action queue is run in the simulation.

[7] A deep copy is described in the Appendix entry "Deep copy" A.2.

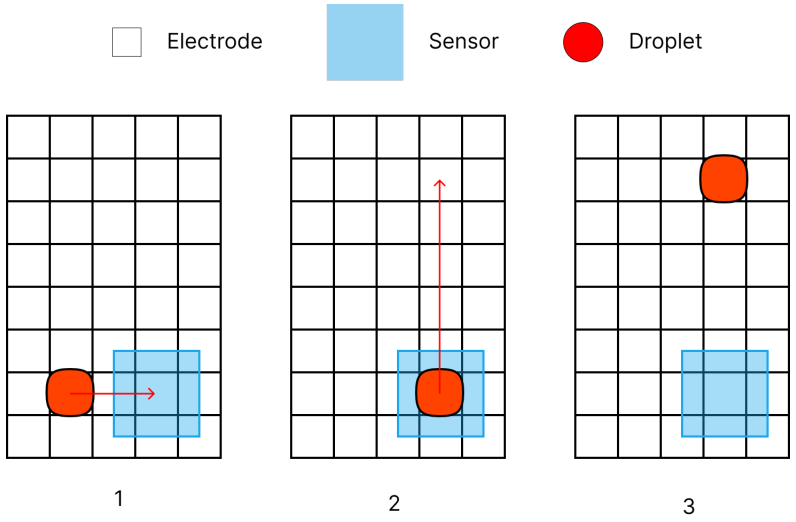

Figure 5.14: 1. The droplet is moving towards the sensor.
2. The sensor reads the colour of the droplet and pushes an action queue to the running action queue.
3. The recently pushed action queue has been executed, and the droplet has moved upwards on the board.

The action queue models contain the logic for adding one or multiple actions to the action queue that is being executed at runtime; the model which is currently in use is **pushActionQueueToOriginalActionQueue**, which intertwines the pushed actions with the simulator action queue. The function traverses the actions in both queues and adds them in chronological order to a new action queue which can then be used to overwrite the action queue of the simulator.

5.6 Simulation Engine Summary

In this chapter, we described the role of the simulation engine, which is to keep track of all the components in the container and decide how the components' models are executed. The part of the chapter describing the functionality of the simulation engine can be divided into three parts the sections "Model Chooser", "Simulator and Time", and "Subscriptions" these three parts comprise the simulation engine itself. This chapter discusses the intricate time functionalities and explains how time-dependent models are handled by dividing big time leaps into smaller ones. Furthermore, the chapter describes how a simplified virtual

machine can interact with the simulation engine. After reading this chapter, one should understand how models are executed and how the simulation engine keeps track of all components and their models, thereby choosing which component should run a specific model at a given time.

Chapter 6

Graphical User Interface

This chapter presents and explains the design and implementation of the Graphical User Interface (GUI) and the methods used for data transfer between the Simulation Engine and GUI.

In Figure 6.1, the final product of the GUI can be seen, and this is also the product that is explained and built throughout this chapter. The GUI is presented in components, allowing for an easier understanding of the process and inner connection. The GUI components are accumulated and summarized at the end of each component, slowly building the end product.

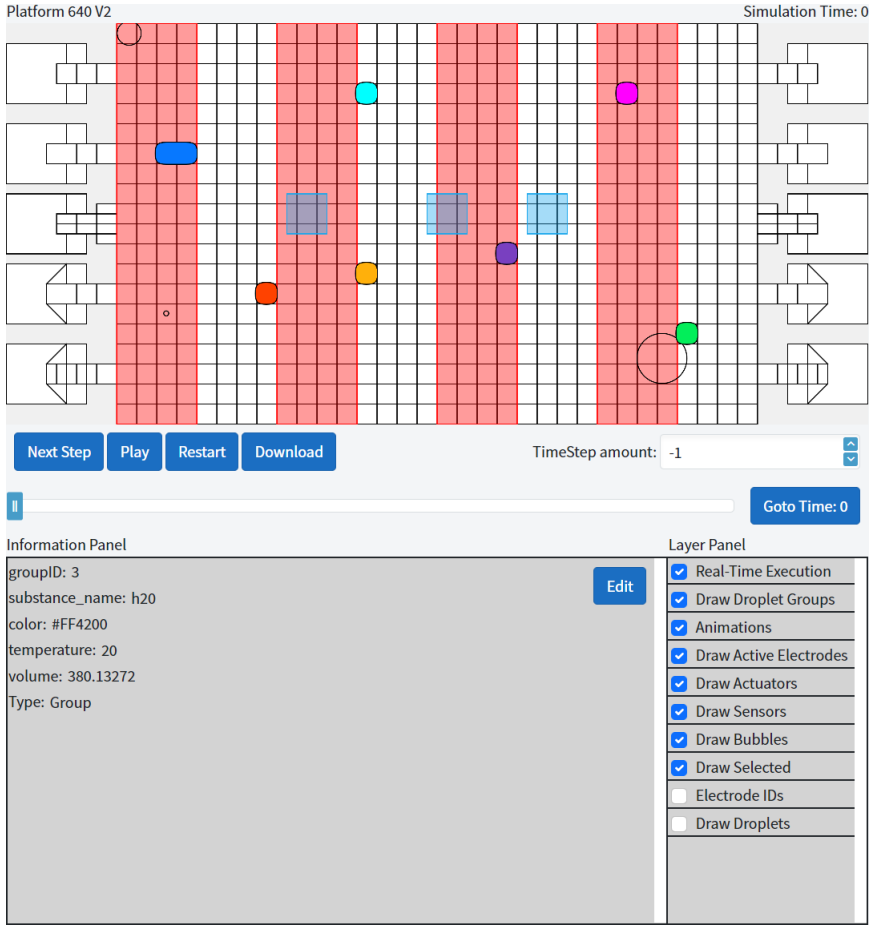

Figure 6.1: Final product of the GUI.

6.1 View

This section presents the view design, overall structure of the GUI and implementation tools used.

6.1.1 Design

The GUI is the visual representation of the Simulation Engine and serves as the only interactive component for the user. It is therefore important to design the GUI in such a way that the user feels fulfilled in regards to functionalities and visuals.

The GUI consists of a frontend and a backend. The frontend contains everything visible to the eye, for example, structure, styling, etc., and the backend contains the logic. The GUI is developed mainly in JavaScript and is integrated into the Blazor/Razor framework.

One of the key design choices regarding the visual composition of the GUI was to achieve a composition which would house functionalities allowing the user to control the Simulation Engine from the GUI, but also be compact, simple and intuitive, providing an easily approachable and some-what recognizable environment for the user. The visual simulation of the DMF platform should be completely recognizable and adhere to the reality observed in the DMF platform. DMF platform behaviour comes from the Simulation Engine; however, the visual representation of the behaviour should be modelled in such a way that it reflects the physical DMF platform.

To accompany the user's need to interact with and control the simulation, we produced a set of functionalities which we deemed necessary for the GUI to contain. The functionalities include a way to manually step through the simulation, play the simulation, interact with the elements of the simulation and download data related to the simulation. In conjunction, these functionalities should provide the user with the necessary control over the simulation. The functionalities are listed and shortly described below.

- Step Functionality
  Allowing the user to manually step through the simulation, displaying one time step at a time.
- Play Functionality
  Allowing for automatic stepping through the simulation.
- Restart Functionality
  Allowing the user to restart the simulation.
- Download Functionality
  Allowing the user to download data from the simulator.
- Edit Functionality
  Allowing the user to edit attributes of elements in the simulation engine.

We found this to be the minimal set of functionalities, providing the desired control. However, this set of functionalities is not a complete set, meaning it should be possible to add other functionalities at a later point in time.

Frontend

We have chosen to build our visual composition around four main containers (boxes), each containing different components that together make up the GUI and provide the desired functionality. The four containers are as follows:

1. The Sketch Panel (Main View Port)
   Contains the actual view of the DMF platform simulation, meaning all elements related to the simulation (electrodes, droplets, etc.).
2. The Control Panel
   Contains various interactable components (buttons, sliders etc.) used to control the execution of the simulation, shown in the sketch panel.
3. The Information Panel
   Used to display information about a select element from the simulation.
4. The Selection Panel
   Contains a set of toggleable components, which are used to turn on or off different properties of the simulation, shown in the sketch panel.

In Figure 6.2 the block design for the GUI is shown. From this block design, we can populate each container with the correct components. With this design, we believe that we are able to achieve our specifications and create a user interface which is both compact, simple and intuitive, providing all the necessary functionalities within a reasonable distance of the main viewport, sketch panel.

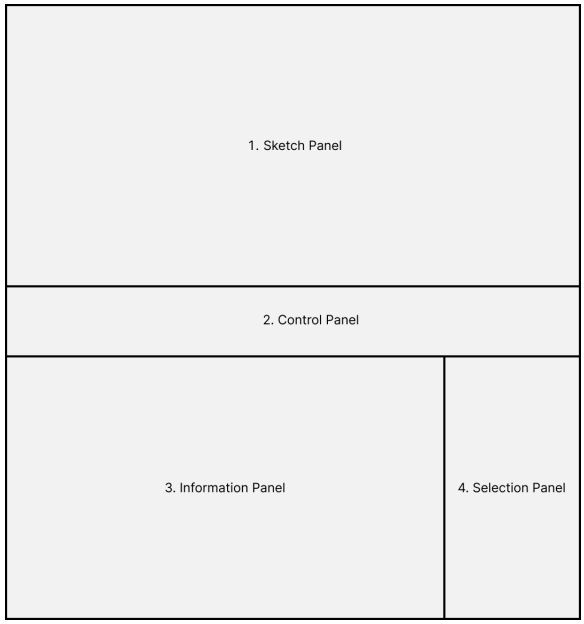

Figure 6.2: Main view components within the GUI.

Backend

The GUI is structured in such a way, that for each of the four containers in the frontend 6.2, we have corresponding managers on the backend. Those managers are responsible for the functionalities that the different view panels need and exchange the necessary information with the associated panels, see Figure 6.3. We also have a broker, which acts as the pipeline between the GUI and the simulator, containing the logic that allows for communication between the two. Since we are using C# and JavaScript, we need a broker in both languages, which are then connected; one could think of those as the ends of the data pipe. The main simulator view is contained within what is referred to as a sketch, which is terminology borrowed from the p5js framework, used to visualize the simulator.

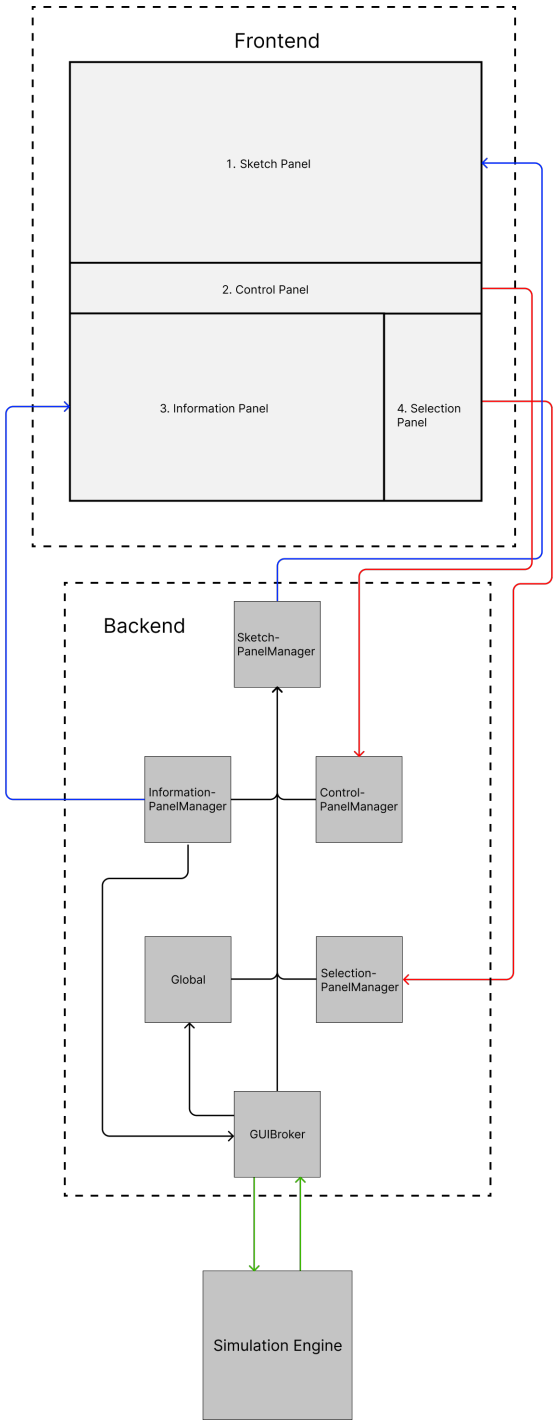

Figure 6.3: Arrows describe the interactions and are colour coded, red is output from the frontend to the backend, blue is input from the backend to the frontend, black is intra GUI connections, and green is interactions between the Simulator Engine and GUI backend.

As seen in Figure 6.3 we have a total of four managers and one broker, namely:

- Information Panel Manager
  Manages the Information Panel on the frontend; it holds information about an element, which can be selected through the Sketch Panel.
- Selection Panel Manager
  Manages the Selection Panel on the frontend; it controls the application of different properties on the Sketch Panel.
- Sketch Panel Manager
  Manages the Sketch Panel on the frontend; it contains all logic related to the visual representation of the DMF platform.
- Control Panel Manager
  Manages the Control Panel on the frontend; the main functionality is the logic behind downloading data from the simulation.
- GUI Broker
  Manages data transfer to and from the Simulator Engine, and it is the only part of the GUI which can transmit and receive data from outside the GUI.

The last component of the backend, which is not included in this list of managers and brokers, is the Global component, which can also be seen in Figure 6.3; this component contains global variables and functions, accessible to all parts of the GUI. Together, these components make up the backend of the GUI.

Data

Since our GUI is separate from our Simulator Engine, we need to re-represent the data from the engine in the GUI. The GUI is developed in JavaScript, meaning that the data structure does not adhere to any primitive data types. This means we can directly deserialize any data from the Simulation Engine into a JavaScript object. The GUI board data structure can be seen in 6.1.

```
type current_time    = float

type platform_name   = string
type platform_type   = int
type platform_id     = int
type sizeX           = int
type sizeY           = int
type Information     = platform_name * platform_type * platform_id *
                       sizeX * sizeY

type name            = string
type ID              = int
type electrodeID     = int
type driverID        = int
type shape           = int
type positionX       = int
type positionY       = int
type status          = int
type corners         = int list
type Electrode       = (name * ID * electrodeID * driverID * shape *
                       positionX * positionY * sizeX * sizeY * status *
                       corners)
type Electrodes      = Electrode list

type substance_name  = string
type volume          = float
type color           = hexidecimal
type temperature     = float
type Droplet         = (name * ID * substance_name * volume * positionX
                       * positionY * sizeX * sizeY * color * temperature
                        * electrodeID)
type Droplets        = Droplet list

type valueActualTemperature     = float
type valueDesiredTemperature    = float
type valuePowerStatus           = int
type actuatorID      = int
type Actuator        = (name * actuatorID * valueActualTemperature *
                       valueDesiredTemperature * valuePowerStatus *
                       positionX * positionY * sizeX * sizeY * color *
                       type)
type Actuators       = Actuator list

type valueRed        = int
type valueGreen      = int
type valueBlue       = int
type sensorID        = int
type Sensor          = (name * sensorID * valueRed * valueGreen *
                       valueBlue * positionX * positionY * sizeX * sizeY
                        * color * type)
type Sensors         = Sensor list

type Bubble          = (name * ID * positionX * positionY * sizeX *
                       sizeY)
```

```
type Bubbles        = Bubble list

/* The container is the main data structure, and it contains
all information needed about the elements */
type Board      = (Information * Electrodes * Droplets * Actuators *
                    Sensors * Bubbles)

/* GUI Specific */
type vertexes       = (int * int) list
type DropletGroups  = (Droplets, vertexes) list
```

Code Snippet 6.1: Shows the data structures and data types, using a descriptive pseudo code.

Code Snippet 6.1 only contains data values used by the GUI; however, the Simulation Engine uses additional values. The additional values can be seen in Section A.3. The additional data values are present within the GUI; however not used by the GUI as per the writing of this thesis.

It is worth noting the data structure DropletGroups. This is a GUI specific data structure, meaning that it is not used by the Simulation Engine. This data structure is used to distinguish the different groupings of droplets, meaning droplets close to one another are considered one droplet.

6.1.2 Implementation

To implement the GUI, and populate the design, we utilize HTML for structuring, CSS/bootstrap for styling, JavaScript to make dynamic updates and provide interactivity for the user, and Blazor/Razor as we should be able to incorporate our design in this specific framework. To populate the Sketch Panel with the visual representation of the DMF simulation, we utilized a framework called p5js [12], which is an extension to JavaScript, that provides a strong visual engine.

When a Razor page is first loaded or updated, the function *OnAfterRender* is called. This function contains a boolean called *firstRender*, which signals if this is the firstRender, or just a render update. By utilizing this boolean, we can initialize our GUI when the Razor page is first loaded. This is especially important if we want to update view elements based on data passed in from the Simulation Engine.

6.2 GUI Broker

This section describes the methods used to transfer data between the Simulator and GUI, carried out by the data exchange manager. See figure 6.3 for GUI structure reference.

6.2.1 Design

The GUI Broker is part of the backend of the GUI and acts as the only connecting link between the Simulation Engine and GUI. The GUI Broker is, therefore, responsible for all data exchange, see figure 6.3. Its main purpose is to exchange DMF platform data so that the GUI can be updated in sync with the simulation engine. The data should be able to be exchanged in real-time, allowing for more realism within the visual representation of the simulation. Furthermore, the GUI Broker also act as a temporary data storage, meaning that it stores the exchanged DMF platform data until new data overwrites the stored data.

```
/* GUI Broker data storage */
type Electrodes     = Electrode list
type Droplets       = Droplet list
type Board          = (Information * Electrodes * Droplets * Actuators
                      * Sensors * Bubbles)
type DropletGroups  = (Droplet list * (int * int)) list

/* GUI Broker functions */
type time           = float
type elementtype    = string
type json           = string

function next_simulator_step: () -> Board
function next_simulator_step_time: time -> Board
function init_board: () -> ()
function update_simulator_container: elementtype * json -> Board
```

Code Snippet 6.2: Describes the structure of the GUI Broker data types and functions, using descriptive pseudo code.

Code Snippet 6.2, displays the data types used by the GUI Broker, as well as some functions which will be responsible for the data exchange. The purpose of the functions are as follows:

- **next_simulator_step()**

Calls on the Simulation Engine to run the next simulator step (next action), resulting in the Simulation Engine returning the updated board data.

- **next_simulator_step_time(time)**
  Takes a time and calls on the Simulator Engines to step with the given time, resulting in the Simulation Engine returning the updated board data.
- **init_board()**
  This function is used to initialize the board when the GUI is initially loaded. The initial data is sent from the Simulation Engine on load. The function does not return anything. However, it does cause some updates on the front end of the GUI, such as displaying the board name as well as setting the dimensions of the board.
- **update_simulator_container(elementtype, json)**
  This function takes a DMF element type, such as an electrode, droplet, etc. and a JSON string representing the corresponding element class. The function is used to update element attributes in the Simulation Engine from the GUI.

Together these functions create the foundation of the data exchange between the GUI and Simulation Engine.

All data exchange is handled in the Razor page called SimulatorPage, where the simulation engine and C# GUI Broker are initialized.

6.2.2 Implementation of Data Exchange

As described in the previous sections, the most important task of the GUI Broker is to exchange data with the Simulator Engine. Since the GUI and Simulator engine are operating in two different programming languages, this task becomes a bit more advanced.

The task can be split up into two sub-tasks by the communication direction, GUI → Simulator Engine and Simulator Engine → GUI.

We will first tackle the easier communicating direction, namely GUI → Simulator Engine (JavaScript → C#). To accomplish this, we can use a built in function called DotNet.InvokeMethod, which has the exact functionality we need. However, we have the choice between an asynchronous version, DotNet.InvokeMethodAsync, or a synchronous version, DotNet.InvokeMethod. Since

our application runs client side without any server functionality, it is, therefore, redundant to use the asynchronous version, and therefore we opt to use the synchronous version. We are now able to call C# methods within JavaScript, and we can also pass arguments by using the following syntax for the method:

DotNet.InvokeMethod(Solution, Method Name, arguments);

The C# methods, called using the InvokeMethod, have to be static and have a JSInvokable tag, signalling that they can be invoked by JavaScript. See the example in Code Snippet 6.3.

The following is an example of the GUI calling the Simulator Engine and telling it to perform the next simulator time step:

```
/* Call: JavaScript (Located in GUI Broker) */
DotNet.InvokeMethod('MicrofluidSimulator', 'JSSimulatorNextStep');

/* Invokes: C# (Located in SimulatorPage) */
[JSInvokable]
public static void JSSimulatorNextStep() {
    // Tell the simulator to perform a time step
    simulator.simulatorStep(-1f); // -1 equals step to the next action

    // Tell the C# GUIBroker to update the board data and send it
    guibroker.update_board(simulator.container);
}
```

Code Snippet 6.3: Shows a call from the GUI to the Simulator Engine.

This gives us the ability to directly call C# methods from JavaScript, given that the methods adhere to the requirements of the DotNet.InvokeMethod function.

We will now tackle the other communication direction, Simulator Engine → GUI (C# → JavaScript), which is a bit more challenging. To accomplish the data transfer, we have to utilize a functionality contained in the Blazor/Razor framework called JS Interop. JS Interop allows us to make use of three different runtimes:

1. IJSRuntime
   Asynchronous invocation of JavaScript functions.
2. IJSInProcessRuntime
   Synchronous invocation of JavaScript functions.

3. IJSUnmarshalledRuntime
   Synchronous, but calls may be dispatched without JSON marshalling.

By using the same arguments as for the other communication direction, we are excluding the asynchronous choice since it is redundant on a purely client based application. This leaves us with the in process runtime and the unmarshalled runtime. The main difference between the two is their execution time, where the unmarshalled runtime is the fastest [13].

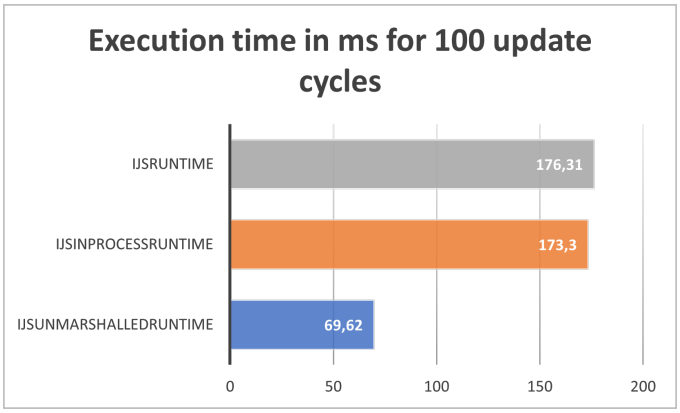

Figure 6.4: Execution time analysis in ms for 100 update cycles.

In the diagram in Figure 6.4, the unmarshalled method seems favourable by a big margin. However, it comes with some drawbacks. The main drawback with the unmarshalled method is that it utilizes mono argument binding in the JavaScript function. This means that we have to use mono methods to convert types manually. The mono methods are not documented and may change in the future, see reference [13]. This means that the unmarshalled method could be unreliable since it could change at some point in the future. As a consequence of this information, we chose only to utilize the unmarshalled method for one case, which is the most frequent data transfer, when the Simulation Engine data transfers DMF platform data to the GUI. In all other cases, we use the in process runtime, which is documented and therefore more reliable. It is worth mentioning that changing from the unmarshalled to the in process runtime is rather straightforward, and should it be the case that the mono methods change, it can then be changed to the in process runtime.

We now have two methods to call JavaScript functions. However, they are not able to see the scope for which our JavaScript functions are located. Both the unmarshalled and the in process runtime methods are only able to see the

”global” scope, meaning we need to provide the JavaScript functions within a ”global” scope. To expose our JavaScript functions to the ”global” scope, we bind the JavaScript functions to the window object, which references the browser's window [14]. The window object is accessible by all view components, and by binding the functions to the window object, we are now able to call JavaScript functions from C#.

We can now extend Code Snippet 6.3, where the GUI calls the Simulator by displaying the call back to the GUI from the Simulator:

```
/* Call: JavaScript (Located in GUI Broker) */
DotNet.InvokeMethod('MicrofluidSimulator', 'JSSimulatorNextStep');

/* Invokes: C# (Located in SimulatorPage) */
[JSInvokable]
public static void JSSimulatorNextStep() {
    // Tell the simulator to perform a time step
    simulator.simulatorStep(-1f); // -1 equals step to the next action

    // Call: GUI
    guibroker.update_board(simulator.container);
}

/* GUI Broker C# function */
public void update_board(Container container) {
    _JSUnmarshalledRuntime.InvokeUnmarshalled<string, string>("
                    update_board", Utf8Json.JsonSerializer.
                    ToJsonString(container));
}

/* JavaScript function called by C# */
window.update_board = (_container_string) => {
    /* Convert JSON string to an object */
    let container_string = BINDING.conv_string(_container_string);
    var board = JSON.parse(container_string);

    /* Update the gui_broker information */
    gui_broker.board = board;
    gui_broker.droplets = board.droplets;
    gui_broker.electrodes = board.electrodes;
    gui_broker.prev_droplet_groups = gui_broker.droplet_groups;
    gui_broker.get_droplet_groups();
}
```

Code Snippet 6.4: Shows a full update cycle.

We have now completed a full data transfer cycle, sending data from the simulation engine to the GUI and back.

6.2.3 Real-time Execution

At this point, we are able to execute a full data transfer cycle between the simulation engine and GUI. However, the data transfer cycle is executed as fast as it is able to, where the only limiting factor is the execution time of the data transfer method and the execution time of serializing/deserializing the data. To achieve real-time execution of the simulation instructions, we have to keep track of the current simulation engine time and the next simulation engine time, and then only instantiate a new data transfer cycle when the difference in time has passed in real time. As an example, consider the scenario where the current simulation engine time is 0 seconds, and the next action is executed on time 10 seconds. Without real-time execution, the execution would happen near instantaneous, and the simulation engine would merely update the current time to 10 seconds, but with real-time execution, the GUI would wait 10 actual seconds and then instantiate the next data transfer cycle. The data sent from the simulation engine and used by the GUI, seen in Code Snippet 6.1, has a `current_time` attribute, which is used to store the current time of the simulation. We can use this attribute to achieve the described real-time execution. By storing the previous current simulation time, when receiving a new data transfer from the simulation engine, we can calculate the difference between the two.

$$\Delta t = t'' - t'$$

t' is the previous, current time, and t'' is the new current time, subtracting the two results in the time difference, Δt. Δt is the time which should be passed before instantiating a new data transfer cycle. To find the real time passed, we can use the JavaScript function `Date.now()`, which returns an integer representing milliseconds elapsed since January 1, 1970. By storing a time instance of `Date.now()` when the GUI receives data from the simulation engine and then finding the difference between the stored time and the current `Date.now()`, we can check when this difference is equal to Δt. We need to ensure that the difference is converted from milliseconds to seconds since Δt is stored in seconds. By checking Δt against the difference in `Date.now()`, we can find when the Δt time has passed.

This approach does, however, raise an issue, it does not account for the execution time which is used to serialize, deserialize, and send the data. As an example, let's look at the scenario where the previous time is 0 seconds and the current time is 0.1 seconds. From Figure 6.4 under the unmarshalled runtime, we can see that one data transfer cycle takes, on average takes, approximately 70 milliseconds or 0.07 seconds, meaning that 70 percent of the actual simulation time is already used to send the data from the simulation engine to the GUI. If the simulation and physical DMF platform were to run simultaneously, the execution time, which is not accounted for, would make the simulation fall

behind the actual platform. At this time, we see two potential ways of tackling this issue:

- **Buffering**
  By executing multiple data transfers and then buffering them on the GUI, keeping X iterations of data, we can eliminate the extra time, since the data has already been transferred.
- **Percentage Speed-up**
  By finding the percentage of the data transfer cycle time, is of the actual simulation time difference Δt, we can speed up the time by a corresponding percentage amount.

The potential solutions to the proposed problem does present their own issues. The **buffering** solution presents the issue where the buffer needs to be recalculated if the simulation engine data changes during the simulation. Another issue with this method is that if the execution time between actions in the simulation engine is significantly small, our buffer would have to include every simulation engine execution, meaning that we essentially first run the simulation, save the data, and then replays the simulation via the GUI.

The **percentage speed-up** solution will only work when Δt is greater than the data transfer time. Otherwise, execution would appear instantaneous in the GUI. As an example, if Δt is 0.04 seconds, the percentage of the data transfer time would be over 100 percent since Δt is less than the data execution time of 0.07 seconds, meaning that the percentage speed-up would be 100 percent, meaning no time should be waited.

None of these solutions are currently implemented but could be implemented. The percentage speed-up solution would be the most straightforward implementation, but the buffering solution would most likely be the better solution.

6.3 Sketch Panel

This section presents and describes the Sketch Panel and the Sketch Panel Manager, as described and displayed in Figure 6.3.

6.3.1 Design

The Sketch Panels' overall design and placement in relation to the rest of the GUI can be seen in Figure6.2, in Figure 6.3, the general placement and connection between the Sketch Panel Manager and Sketch Panel can be seen. In Figure 6.3we can see that the Sketch Panel Manager is a central part of the backend; it essentially acts as the main component for which all other backend components are called. This is because the Sketch Panel Manager is in control of the Simulation representation, which is essentially the main purpose of the GUI. This also means that the Sketch Panel is the main part of the frontend of the GUI since this is the container for which the actual simulation of the DMF platform will be shown.

The Sketch Panel can be split into two parts, a header containing the DMF platform board name and the current simulation time and a canvas housing the p5js sketch. Within this canvas, the p5js visual framework is utilized to display the digital representation of the DMF platform. The p5js framework is contained in the Sketch Panel Manager, which updates the canvas dynamically.

6.3.2 Implementation

The structure of the Sketch Panel is built up by HTML code, the HTML code can be seen in Code Snippet6.5.

```
<div id="sketchPanel">
    <div id="simulatorView">
        <div id="simulatorHeader">
            /* Will be updated with the correct board name */
            <span>BOARD NAME</span>
            <div>
                /* Will display the current simulation time */
                <span>Simulation Time: </span>
                <span id="simulatorTime"></span>
            </div>
        </div>
        /*
         * The div with id 'container' is where
         * the p5js canvas will be programmatically added later
         */
        <div id='container'></div>
    </div>
</div>
```

Code Snippet 6.5: Shows the HTML behind the Sketch Panel.

The HTML code is located within the Razor page called SimulatorPage.

The HTML structure contains different elements, which are initialized during the setup of the Sketch Panel Manager; this happens when the page is first loaded. The elements are as follows:

- Board Name - Updated to fit the DMF platform name contained in the board data.
- Current Simulation Time - Updated dynamically.
- p5js Sketch container - Updated dynamically but initiated to the initial simulation data at the start of the simulation.

All of these elements can also be found in Code Snippet 6.5.

We load and populate these elements during the initialization of the GUI. At this point, our GUI Broker has already received its first batch of Simulation Engine data, meaning we have access to the initial simulation positions at time 0. We can now populate the Board Name to be the one which is passed in with the data and corresponds to the actual DMF platform name and the simulation time, which can now be set to 0.

To initialize the p5js Sketch Container, we have to create a new p5js instance, which can be done as seen in the Code Snippet 6.6:

```
/**
 * The function setp5 initializes the p5js instance (the sketch),
 * and binds it to the element with id 'container'.
 */
window.setp5 = () => {
    new p5(sketch, window.document.getElementById('container'));
    return true;
};
```

Code Snippet 6.6: Shows the p5js setup.

The function is located in JavaScript; however, it is bound to the window scope, meaning it is global and accessible by the whole GUI. This furthermore means that the function can be called directly from the Razor page (see section 6.2 for data exchange) upon the first Razor page render.

By now, we have a Sketch Panel with a header containing initial information about the DMF platform simulation and a canvas housing a p5js instance; however, nothing has been drawn onto the canvas yet. To do this, we have to take a look at the Sketch Panel Manager.

6.3.3 Sketch Panel Manager

This section describes the core applications of the Sketch Panel Manager, showing how we utilize the p5js framework to visually represent the DMF platform simulation as close to reality as possible.

Design

The Sketch Panel Manager manages the Sketch Panel, utilizing the p5js framework to visually represent the DMF platform simulation. The Sketch Panel Manager draws the DMF platform information onto the Sketch Panel canvas, and the information comes from the Simulation Engine. The Sketch Panel Manager should be able to draw DMF platform elements, such as electrodes, droplets etc. and also components, such as sensors and actuators, onto the canvas. Furthermore, the Sketch Panel Manager should supply animation logic to the droplets. The simulation elements should be represented in such a way that they are recognizable and distinguishable.

We have designed the following types and functions, to accompany the design for the Sketch Panel Manager.

```
1  /* p5js functions */
2  function setup: () -> ()
3  function draw: () -> ()
4
5  /* Initialize function */
6  function init_board: () -> ()
7
8  /* Draw functions */
9  function draw_electrode:        electrode -> ()
10 function draw_droplet:          droplet -> ()
11 function draw_droplet_group:    int -> ()
12 function draw_actuator:         actuator -> ()
13 function draw_sensor:           sensor -> ()
14 function draw_bubble:           bubble -> ()
15
16 /* Animation functions */
17 function animate_droplet_group: droplet -> ()
```

Code Snippet 6.7: Shows functions for the Sketch Panel Manager, using a descriptive pseudo code.

The input and output of the Sketch Panel Manager are described in Code Snippet 6.7. We continue to present a short explanation of their functionality.

p5js functions

- **setup()**
  The setup function is part of the p5js framework. It is called when the p5js instance, also referred to as the sketch, is first initialized; this function is used to setup the p5js instance canvas, which is what will be displayed in the Sketch Panel canvas.

- **draw()**
  The draw function is part of the p5js framework. The function is called every frame of the sketch and is used to update the sketch canvas elements dynamically while the sketch is running.

Initialize function

- **init_board()**
  The init_board function is used to initialize various properties, such as the initial board elements (electrodes, droplets, actuators, sensors etc.).

Draw functions

- **draw_electrode(electrode)**
  Controls the drawing of the electrodes onto the Sketch Panel canvas, utilizing the p5js framework. Most electrodes are pure squares; however, some electrode has other shapes, and therefore this function should also support arbitrary polygonal electrode shapes.

- **draw_droplet(droplet)**
  Controls the drawing of individual droplets; the droplets are represented as circles, where the size correlates to their volume. The droplets have unique colours, which are contained from the data passed by the Simulation Engine.

- **draw_droplet_group(int)**
  Controls the drawing of droplet groupings, meaning individual droplets, which are close enough to be considered one droplet, should be drawn as one droplet. The groups will be drawn based on the positions of the individual droplets which make up the droplet groups.

- **draw_actuator(actuator)**
  Controls the drawing of actuators, e.g. heaters, distinguishable by colour and shape within the simulation contained in the Sketch Panel.

- **draw_sensor(sensor)**
  Controls the drawing of sensors, e.g. temperature and RGB sensors, distinguishable by colour and shape within the simulation contained in the Sketch Panel.

- **draw_bubble(bubble)**
  Controls the drawing of bubbles, the bubbles are represented by a circle.

Animation function

- **animate_droplet_group(droplet)**
  Controls procedural animation applied to every droplet group, which allows for smooth and realistic movement of the droplet groups.

Every function described above utilizes simulation DMF platform data, which is sent directly from the Simulation Engine to the GUI, as presented in Section 6.2.

6.3.4 Drawing Simulation Elements

This section will describe how the Sketch Panel Manager uses the p5js framework to draw simulation elements (electrodes, actuators, sensors, droplets) onto the Sketch Panel canvas.

Design

In order to preserve distinguishability, some elements, specifically actuators and sensors, deviate from what is observed in reality. By allowing for this deviation, we are able to visually represent the elements in such a way that the user can easily distinguish between the elements.

The visualization of the different elements is as the following:

- Electrodes - Arbitrary polygons, in most cases squares, with a white background and black outline.
- Actuators - Rectangles with a transparent red background and a red outline.
- Sensors - Squares with a transparent blue background and blue outline.
- Droplets - Circles with a black outline and background corresponding to the colour given by the data from the Simulator Engine.

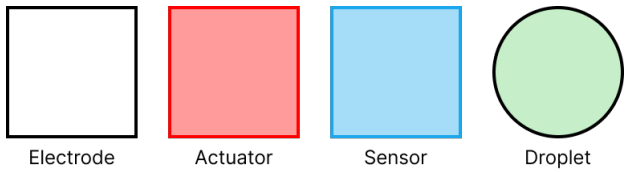

Figure 6.5: Design overview of the different simulation elements.

As described in the Simulation Engine chapter 5, an electrode can either be turned on or off, and this should be depicted in the GUI. To distinguish between an off electrode and an on electrode, the on electrode will have a red background instead of the white background, as seen in Figure 6.5.

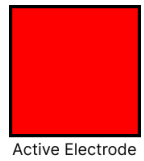

Figure 6.6: Shows an active electrode.

All elements are drawn by the data passed to the GUI from the Simulation Engine, see section 6.2, this data includes positional data and shape data.

Some of the elements drawn onto the canvas, such as electrodes, actuators and sensors, are static, meaning after they are drawn once, they will visually stay the same for the rest of the simulation. Therefore it is redundant to keep recalculating and drawing these static elements every frame of the simulation. To solve this, we can introduce layers, which we can think of as an image. We can then draw the static elements onto the layer once when the GUI is initially loaded and then merely redraw the layer as an image for every update.

Implementation

To draw the elements onto the Sketch Panel canvas, we utilize the p5js framework. Within this framework, there are built-in draw calls for generic geometric shapes, such as rectangles and ellipses. We can use these draw calls to draw the square electrodes, actuators, sensors and droplets.

Drawing Electrodes

To tackle the case of drawing the electrodes, we first have to distinguish between square electrodes and polygonal electrodes. To distinguish between the two, we can use a variable stored within the electrode data type, called shape, see Code Snippet 6.1. The variable shape is an integer that is either 0 or 1, where 0 indicates that the electrode is a square shape and 1 that the electrode is a polygonal shape. By checking this variable's value, we can for each electrode see if it's a square shape or polygonal shape. Square electrodes are rather straightforward since we can use the rectangle draw call to draw the square representing the electrode.

The draw call for drawing square electrodes can be seen in Code Snippet 6.8.

```
1  function draw_electrode() {
2      for (let i = 0; i < gui_broker.electrodes.length; i++) {
3          let electrode = gui_broker.electrodes[i];
4
5          // Border Color
6          layer_electrode.stroke(draw_config.electrode.borderColor);
7
8          // Border Width
9          layer_electrode.strokeWeight(draw_config.electrode.borderWidth)
               ;
10
11         // Fills the Background
12         layer_electrode.fill(draw_config.electrode.backgroundColor);
13
14         // Check the electrode shape
15         if (electrode.shape == 1) {
16             draw_polygon_electrode(electrode.positionX, electrode.
               positionY, electrode.corners);
17         } else {
18             // Using rect to draw a rectangle
19             layer_electrode.rect(electrode.positionX, electrode.
               positionY, electrode.sizeX, electrode.sizeY);
20         }
21     }
22 }
```

Code Snippet 6.8: Function responsible for handling the drawing of square shaped electrodes onto a layer.

In Code Snippet 6.8, on line 19, the call to the p5js rect function can be seen, which draws the actual square onto the Sketch Panel canvas. On lines 6, 9 and 12, respectively, the styling of the electrode is defined; here, stroke is equivalent to the border colour, strokeWeight is equivalent to the border width and fill

is equivalent to the background. All of the styling calls are part of the p5js framework.

Tackling the case of drawing polygonal electrode shapes, we need to be able to support arbitrary shapes, to do this we can utilize another p5js functionality called shapes. This functionality allows us to begin a shape add the necessary vertexes and then end the shape.

```
function draw_polygon_electrode(posX, posY, corners) {
    // Begin the polygon shape
    layer_electrode.beginShape();

    // Create a shape with vertexes corresponding to the corners array
    for (let i = 0; i < corners.length; i++) {
        layer_electrode.vertex(posX + corners[i][0] + 0.5, posY +
            corners[i][1] + 0.5);
    }

    // End the polygon shape
    layer_electrode.endShape(layer_electrode_shape.CLOSE);
}
```

Code Snippet 6.9: Function responsible for handling the drawing of polygonal shaped electrodes onto a layer.

We are now able to draw both square and polygonal electrodes onto a layer. To draw the electrode layer, we can merely use the p5js functionality *image(layer)* to draw the layer as an image onto the Sketch Panel canvas. By running the draw calls, seen in Code Snippet 6.8 and 6.9, on the data passed to the GUI from the Simulator Engine, Figure 6.7 shows the canvas with the electrodes drawn.

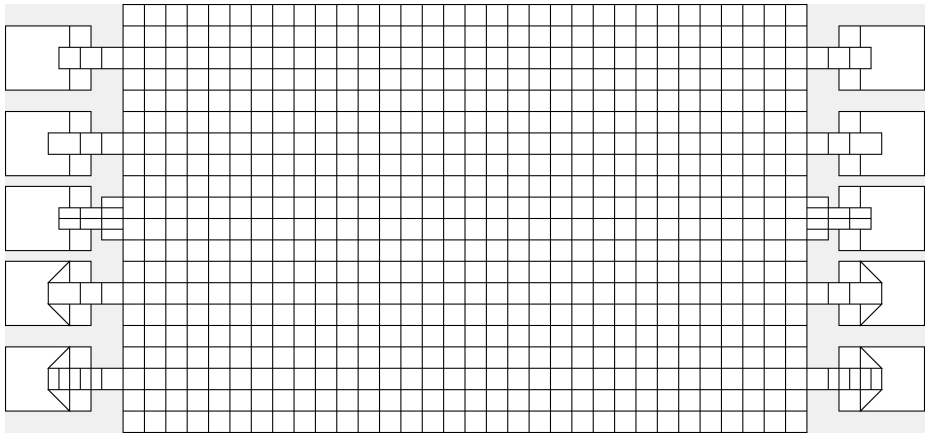

Figure 6.7: DMF platform board drawn onto the Sketch Panel canvas with all electrodes drawn.

We now have all electrodes drawn onto a layer since the shapes are static; however, if an electrode is active, we want to represent it with a red background, as described in the design section. To do this, we can reuse the draw calls from Code Snippet 6.8 and 6.9, but skipping all non-active electrodes, meaning those with the status value set to 0, and redrawing all active electrodes with the fill styling set to red. The active electrodes are drawn on top of the image containing the electrode shapes, seen in Figure 6.8.

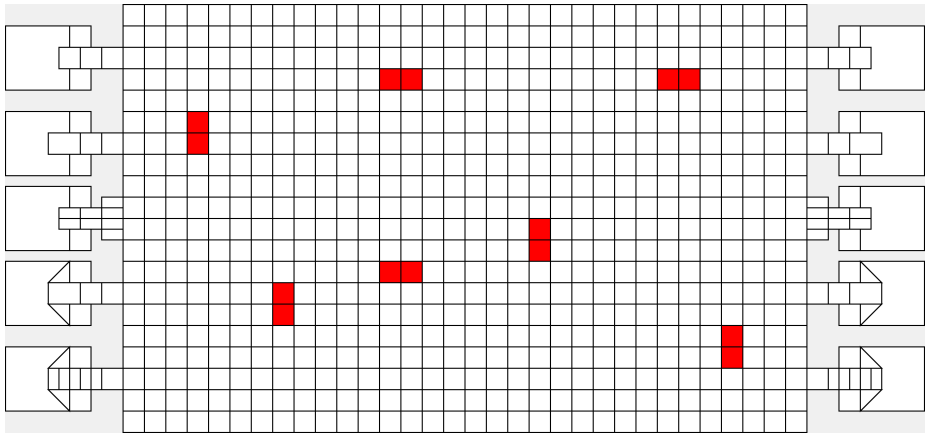

Figure 6.8: DMF platform board drawn onto the Sketch Panel canvas with active electrodes drawn.

Drawing Actuators and Sensors

Next, we tackle the task of drawing actuators and sensors; as for the case of the electrodes, these elements are also static, meaning they both will be drawn onto layers once when the GUI is initialized and thereafter drawn using the p5js *image(layer)* functionality. As to drawing the actuators and sensors onto the layers, we utilize the p5js *rect* draw function since both of these elements are represented as rectangles. Figure 6.9 shows an example of three actuators and two sensors.

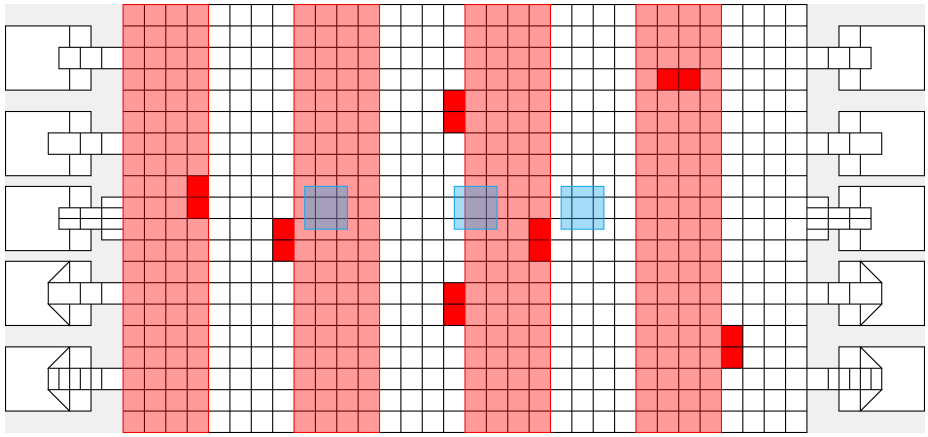

Figure 6.9: DMF platform board drawn onto the Sketch Panel canvas with actuators (red) and sensors (blue).

Drawing Droplets

Continuing, we tackle the task of drawing droplets, which are dynamic elements, meaning they will not stay in the same position during the whole of the simulation. To draw the droplets we use the p5js function *ellipse*, which allows us to draw circles.

```
// Function to draw droplets
function draw_droplet() {
    for (let i = 0; i < gui_broker.droplets.length; i++) {
        let droplet = gui_broker.droplets[i];
        p.fill(droplet.color);
        p.stroke(draw_config.droplet.borderColor);
        p.strokeWeight(draw_config.droplet.borderWidth);
        p.ellipse(droplet.positionX, droplet.positionY, droplet.sizeX,
            droplet.sizeY);
    }
}
```

Code Snippet 6.10: Function responsible for handling the drawing of Droplets.

As with the case of the electrodes, we utilize p5js to style the droplets, this includes the background colour, border colour and border thickness. After a call to the droplet draw function, the following is drawn onto the Sketch Panel canvas.

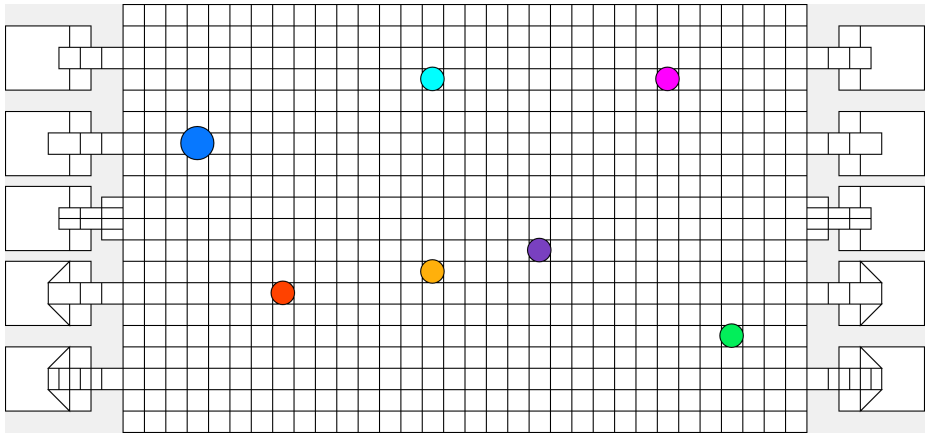

Figure 6.10: Droplets drawn onto the Sketch Panel canvas.

By this point, we are able to draw all elements from the Simulation Engine, and can now represent the full image of the Simulation Engine's data. However, this image is by itself not very realistic looking, since the Simulation Engine represents groups of droplets as multiple individual droplets, located next to one another.

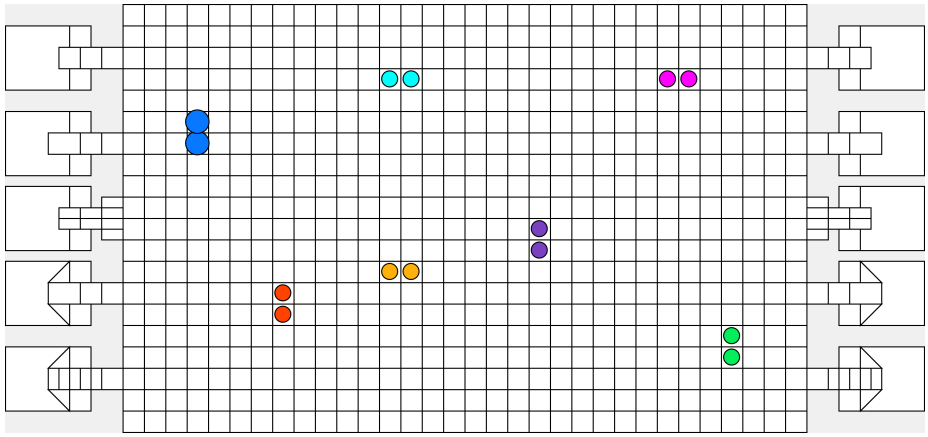

Figure 6.11: Droplets drawn as represented in the Simulator Engine data.

This is not very realistic according to how the droplets look in the actual DMF platform. To achieve realistic looking droplet groupings, we have to alter how the Sketch Panel Manager draws droplet groups onto the Sketch Panel canvas.

6.3.5 Droplet Grouping

This section will describe how the GUI draws droplet groups, the drawing of droplet groups is necessary to allow for more realistic and recognizable droplet compositions and shapes.

Design

When tasked to create visually realistic droplets, we have to reconsider how we visually represent the droplet groups. Droplet groups are considered as individual droplets located directly besides one another. The following image shows figure 6.11 where there is inserted a red bounding box around every multiple of droplets considered a group of droplets.

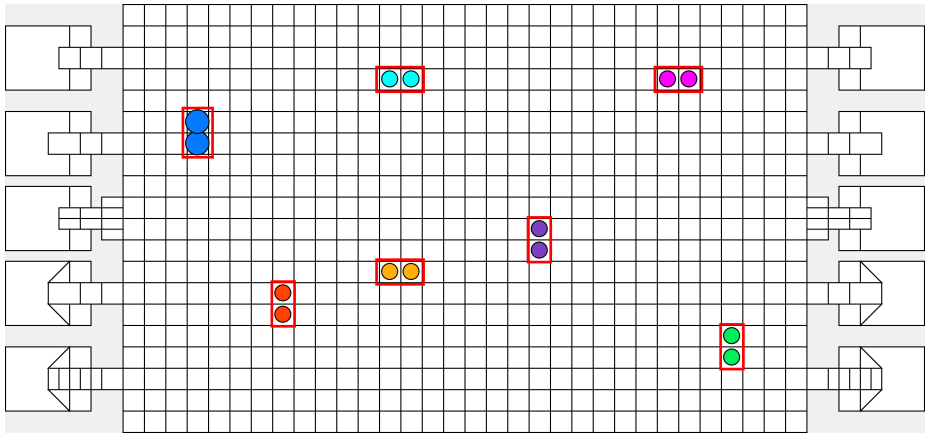

Figure 6.12: Droplet groups shown by a red bounding box around the individual droplets.

The droplets considered within the same droplet group has a groupID equivalent to that of the groups, this groupID is given by the Simulator Engine and passed onto the GUI through the DMF platform data exhange, described in section 6.2.

To emulate the correct droplet grouping behaviour, we want to draw all droplets with the same groupID together as one droplet, instead of as multiple individual droplets. To achieve this, we have to design a custom traversal algorithm.

Designing the Grouping Algorithm

To draw the multiple individual droplets as one droplet, we first have to get data points enclosing the individual droplets within one final shape. This can be achieved by traversing the circumference of the droplet groups gathering points at every corner of the circumference. This traversal should however be done in the same direction around the droplet groups, such that the resulting set of points are ordered, and is essentially a list of vertices, from which a shape can be drawn. By utilizing the positional and sizing properties within the droplet data type, we can find the four points, which creates a bounding square around the droplet.

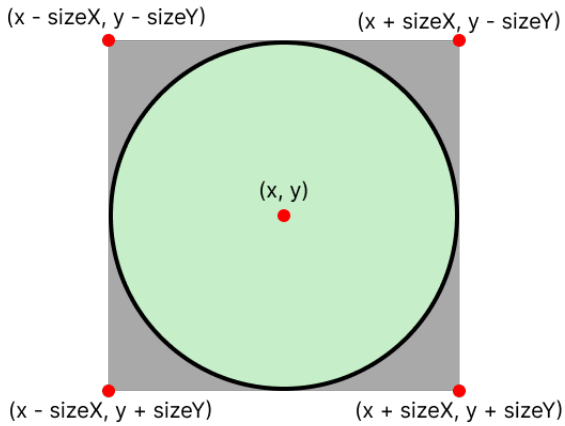

Figure 6.13: Droplet with the four points, which describe the bounding square around a droplet.

These four points creates the basis of the data, and by running the algorithm on the set of droplets in the current group, we can gather the correlating bounding point, to the current corner found in the circumference. However, we are still facing two issues, firstly, when introducing multiple droplets within a group, we see that the bounding points for each of the droplets overlap with one another.

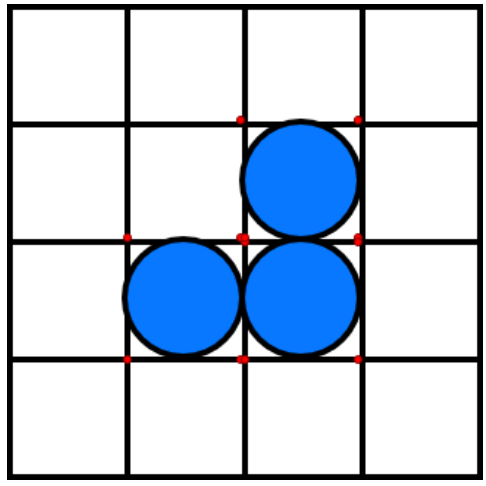

Figure 6.14: Bounding droplet points in a L shaped droplet group.

Secondly, we still have to figure out how we can traverse this set of droplets in

a way that makes the final points ordered so that they can be used as a set of vertices.

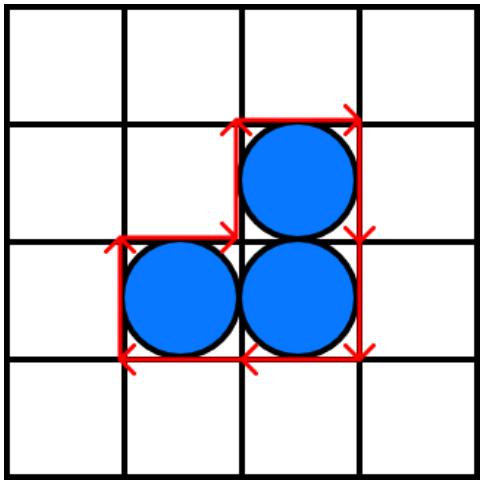

Figure 6.15: Traversal, in order, on the droplet points in an L-shaped droplet group.

The first issue does not present the most significant task, we can be selective in how we choose to gather points, making sure that we only gather one point, even though multiple points can be present in an overlap. The second issue is a bit more complex.

To traverse the droplet group in an ordered direction, we start by selecting a droplet which is missing at least one neighbour. By doing this, we assure that at least one point on the given droplet is part of the circumference of the final shape. We then select one of the points, on the side with no neighbour, as our starting point since we know this starting point to be part of the circumference. From the starting point, we then check to see if there is any droplet (neighbour), within the same droplet group, in four different directions. This can be seen in Figure 6.16.

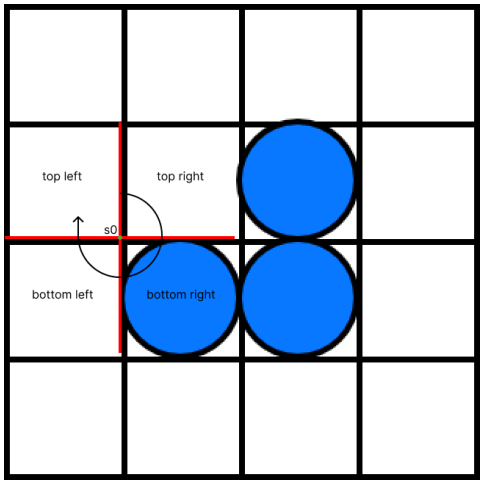

Figure 6.16: Method used for finding the next point while traversing droplet groups.

Figure 6.16 shows the method we use for traversing the droplet group circumference here s0 is our chosen starter point. From s0 we create a type of coordinate system, where for each quadrant, we check to see if the quadrant is empty, meaning containing no droplet, which is neighbour of the current droplet (bottom right), and then check if any of the non-diagonal fields contains a droplet. For s0, the case where the top right is empty and the bottom right is not empty is true, meaning the next valid point is to the right. We check for the cases in the direction of the arrow depicted in the figure, which allows us to traverse the droplet group in the same direction, resulting in an ordered list of points.

With information about the neighbours, we can find the next point on the circumference and add the precise point to a list of the final vertices. By choosing the next point in a clockwise manner, we can assure that the direction is held, as seen in Figure 6.15. We now have a set of vertices for which we can draw a shape that encloses the entire group of droplets. The last step needed for creating a droplet-like structure of the droplet groups is to round the corners of the shape. After these steps, the algorithm should, in theory, be able to draw what is seen in Figure 6.17 onto the Sketch Panel canvas.

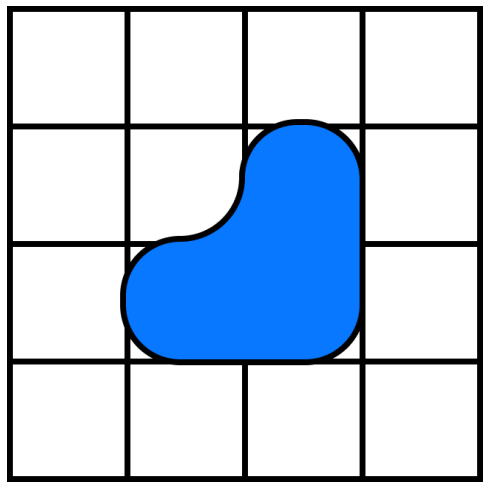

Figure 6.17: Desired result of the algorithm.

Figure 6.17 is not a product of the algorithm, but an estimation of the desired result of the algorithm.

Implementation

To implement the algorithm, we will start by presenting the pseudo code for the algorithm, this will be done in Code Snippet 6.11.

```
1  /* Neighbouring droplet syntax: n_"position of neighbour" */
2
3  foreach dropletGroup
4      initialize vertices list
5      choose starting droplet
6      choose starting point
7      find all neighbouring droplets for the given point
8
9      /* Check for valid traversal points */
10     if n_top_right is null and n_bottom_right is not null
11         /* Point to the right is valid */
12         if current point is convex or concave
13             push corner point to vertices list
14
15         if current point is not contained in vertices list
16             push current point to vertices list
17
18         /* The content of the following else if statements are
               identical to this one, and will be represented by "..." */
19
```

```
20      else if n_bottom_right is null and n_bottom_left is not null
21          /* Point to downwards is valid */
22          ...
23
24      else if n_bottom_left is null and n_top_left is not null
25          /* Point to the left is valid */
26          ...
27
28      else if n_top_left is null and n_top_right is not null
29          /* Point to upwards is valid */
30          ...
31
32
33      draw shape from vertices list
```

Code Snippet 6.11: Pseudo code of the algorithm responsible for traversing the droplet groups and gathering circumference points.

Running the algorithm from Code Snippet 6.11, it will result in the output seen in Figure 6.18.

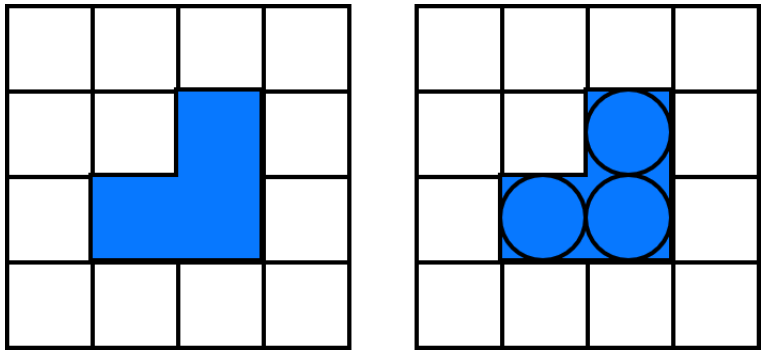

Figure 6.18: Result of the algorithm in Code Snippet 6.11. Showing both the enclosing shape without the individual droplets, and with the individual droplets.

This output is a step in the correct direction, and we are now able to generate the enclosing shape, however, as described in the design section, this is not the desired end result. Droplets have rounded edges, however, this algorithm produces shapes with sharp edges. To get the desired output, we have to round the edges of the shape produced by the algorithm.

To round the corners, we utilize an existing algorithm introduced in reference [15]. The algorithm takes a list of vertices and a radius as input and produces

a shape based on the list of vertices, with corners rounded by the radius. The algorithm uses bezier curves to approximate arches, the arches will be calculated between two points, with a curvature corresponding to the radius. By introducing this rounding algorithm into our algorithm, we get the output seen in Figure 6.19.

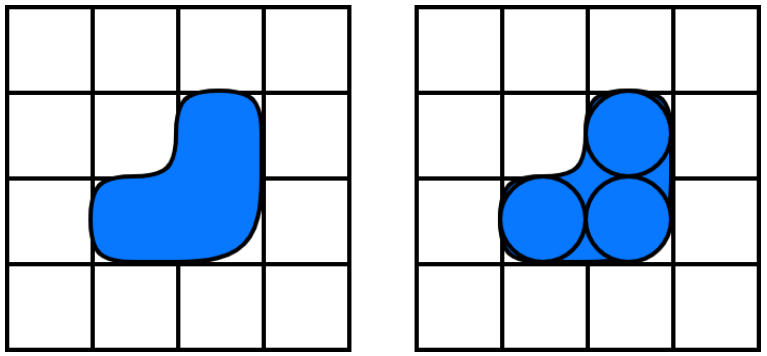

Figure 6.19: Result of the algorithm in code snippet 6.11, with rounded corners. Showing both the final shape without the individual droplets, and with the individual droplets.

With the combination of the two algorithms, we are now able to draw complex droplet group shapes. However, a limitation of this method is that it does not support hollow shapes, meaning shapes with holes.

In Figure 6.20 a comparison between the actual DMF platform and the shape generated by the Sketch Panel manager is presented, and compared by overlaying one over the other. From the comparison, we can see that the simulation produces a shape that is very similar to the one observed in the actual DMF platform.

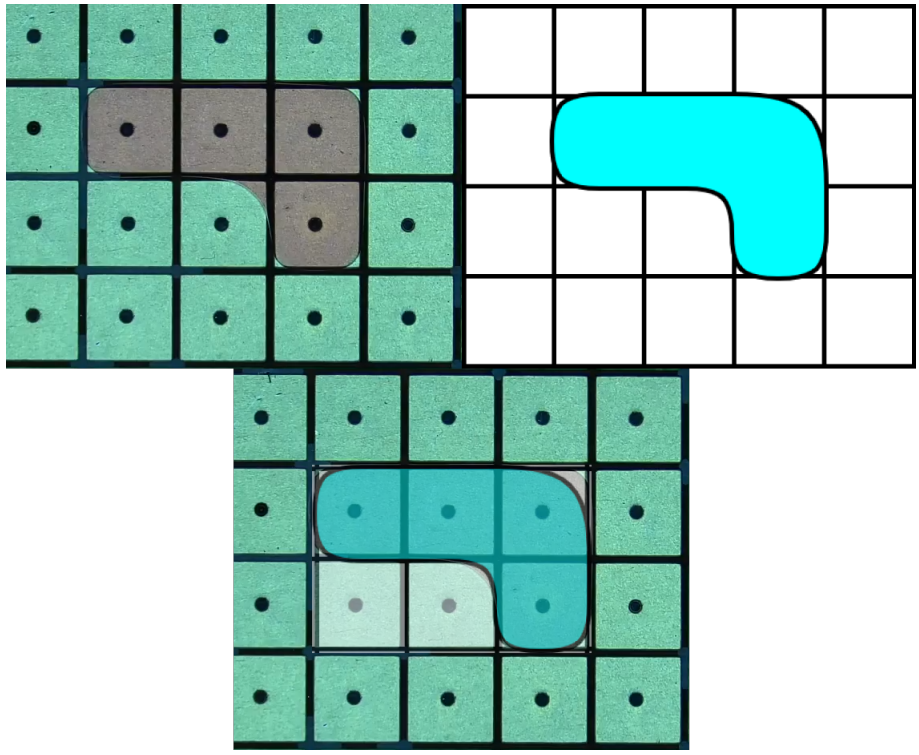

Figure 6.20: Actual DMF platform L-shaped droplet, simulated L-shaped droplet, and an overlay of the two.

6.3.6 Procedural Animations

At this point, we have introduced droplet groups as static shapes, allowing for realistic looking droplets. This section will present the design and implementation of procedural animations, allowing for realistic and dynamic movement, merging and splitting of droplet groups, as seen in the physical DMF platform.

Before beginning this section, we will shortly define the term procedural animation. Procedural animation is a computer animation, where the animation is automatically generated in real-time.

We chose to create procedural animations instead of classic animations since classic animations require predefined animations for every possible configuration of the droplets within the droplet group. On the contrary, procedural

animations only require a set of rules and can be used on every configuration, that follows this set of rules. With this information, it should be quite clear that procedural animations are favourable since it would require a quite large amount of predefined animations to account for every possible animation.

Design of Procedural Animations

The procedural animation should provide realistic movement, merging and splitting of droplet groups, as observed on the physical DMF platform.

A procedural animation model can be designed in many ways, we have chosen to implement an interpolation model. This model works by interpolating between two shapes, one being the initial shape and the other being the final shape. In the GUI, the shapes relate to the droplet group vertices lists, which we described in the previous sections. We chose the initial shape to be the list of droplet group vertices for the current droplet groups and the final shape to be the list of vertices calculated for the droplet groups which are sent from the Simulation Engine. Since we now have two lists of vertices, we can interpolate between the two. However, we first have to figure out which point pairs should be interpolated. To accomplish this, we compare the two lists of vertices and find the points which are in the initial shape and not in the final shape. For each point in the initial shape and not in the final shape, we can find the closest point in the final shape and choose this point to be the one we interpolate to. This model works for droplet groups that are connected between each simulation step since they would have the same groupID. The model would not work for splitting of droplets since when a group splits, one part will keep the groupID, and the other part would get a newly generated groupID. Therefore we have to find the cases where a group splits into multiple subgroups and let our interpolation model run on those cases as well.

Implementation of Procedural Animations

Most of the work has already been done in the previous sections, where we described how we draw droplet groups. From those sections, we are left with all the information we need to implement the interpolation model for procedural animation. Firstly, we need to store both our initial shapes and final shapes, our GUI Broker only acts as temporary storage, meaning that the droplet group data, including the vertices lists, will be overwritten when we receive new data from the Simulation Engine. To save the initial shapes, which are the vertices lists from the current droplet groups, we can create a new temporary storage

within the GUI Broker, which stores the current droplet group data as previous droplet groups since the incoming droplet groups will become the current groups. This way, we are both storing the initial vertices lists for the droplet groups and the final vertices lists.

We can now compare the vertices from the previous droplet groups and the incoming (current) droplet groups and find the vertices, which should be used for the interpolation. We do this by finding the difference between the previous droplet group and the current droplet group. The next step is to find the points that should be interpolated to, meaning that for each point in the comparison difference, we need to find a corresponding point in the current droplet group vertices list. We do this by finding the point, in the current droplet group vertices list, closest to the point in the comparison difference.

```
/* Previous group ID, prevGroupID, is optional */
function getGroupVertexPairsForLerp(groupID, prevGroupID) {
    let cur_group = gui_broker.droplet_groups[groupID];

    if (typeof prevGroupID == "undefined") { prevGroupID = groupID }
    let prev_group = gui_broker.prev_droplet_groups[prevGroupID];

    if (typeof cur_group == "undefined" ||
        typeof prev_group == "undefined") { return; }

    let vertices_to_lerp = getGroupVerticesForLerp(groupID,
                           prevGroupID);
    let vertex_pairs = []; /* [[[x1,x2],[y1,y2]] ... ] */

    /* Find closest point in current group */
    for (i in vertices_to_lerp) {
        let min_dist = [9999, "point"];
        for (j in cur_group.vertices) {
            let dist = p.dist(vertices_to_lerp[i][0],
                              vertices_to_lerp[i][1],
                              cur_group.vertices[j][0],
                              cur_group.vertices[j][1]);

            if (dist < min_dist[0]) {
                min_dist = [dist, cur_group.vertices[j]];
            }
        }
        vertex_pairs.push([[vertices_to_lerp[i][0], min_dist[1][0]],
                           [vertices_to_lerp[i][1], min_dist[1][1]]]);

        vertices_to_lerp.push([min_dist[1][0], min_dist[1][1]]);
    }
    return vertex_pairs;
}
```

Code Snippet 6.12: Function responsible for finding the point pairs, for which an interpolation can be carried out on.

The function in Code Snippet 6.12 returns an array of vertex pairs, with the type `[[[x1, x2], [y1, y2]], ...]`, these point pairs are the ones which will be interpolated between. The last step of the interpolation model is to carry out the actual interpolation, going from the previous point to the new point, from x1 to x2 and y1 to y2. To interpolate, we use the lerp functionality provided by the p5js framework, this function allows us to get a percentage value between two points. Increasing the percentage amount incrementally, from 0 to 1, will allow us to interpolate between the initial point and the final point. We do this for every entry in the array of vertex pairs.

```
/* Previous group ID, prevGroupID, is optional */
function lerpGroupVertices(groupID, amount, prevGroupID) {
  find the cur_group
  find the prev_group

  if prev_group is not undefined {
    /* Using getGroupVertexPairsForLerp */
    find vertex pairs for prev_group
    find vertex pairs for cur_group

    for each entry in current vertex pairs {
      lerp vertex X pairs
      lerp vertex Y pairs

      for each entry in previous vertex pairs {
        if the cur_group vertices should be lerped {
          update the corresponding vertex of the group
        }
      }
    }
  }

  draw the droplet group
}
```

Code Snippet 6.13: Pseudo code of lerpGroupVertices function, responsible for lerping vertices.

In Code Snippet 6.13 we can see the function responsible for handling the lerping (interpolating) between the shapes. The function works by taking the vertex pairs, calculated in Code Snippet 6.12, and lerping from the initial point to the corresponding point in the current droplet group. It does this by a given amount, related to a percentage from 0 to 1, and finds the corresponding value between the vertex pairs. We now have an interpolation model that works for droplet groups containing the same groupID, however, as described in the related design section, we also want to be able to animate splitting, where a new group will be created as a result, and thereby have a different groupID.

By finding the new droplet groups and comparing the individual droplets to the previous droplet groups, we can find correlations between individual droplets since at least one droplet in the new droplet group would be present in the old droplet group. The function used to find the new groups that have been split from a present group can be seen in Code Snippet 6.14.

```
function findGroupSplit(groupID) {
    let cur_group = gui_broker.droplet_groups[groupID];
    let prev_group = gui_broker.prev_droplet_groups[groupID];
    if (typeof cur_group == "undefined" || typeof prev_group == "
            undefined") { return; }

    if (!(Object.keys(gui_broker.prev_droplet_groups).length < Object.
            keys(gui_broker.droplet_groups).length)) { return; }

    let possible_split_droplets = [];
    for (i in prev_group) {
        for (j in cur_group) {

            if (prev_group[i].ID != cur_group[j].ID) {
                possible_split_droplets.push(prev_group[i].ID);
            }
        }
    }

    let new_groups = [];
    for (i in possible_split_droplets) {
        for (j in gui_broker.droplets) {
            if (possible_split_droplets[i] == gui_broker.droplets[j].ID
            && gui_broker.droplets[j].group != groupID) {
                if (!new_groups.includes(gui_broker.droplets[j].group))
            { new_groups.push(gui_broker.droplets[j].group); }
            }
        }
    }

    return new_groups;
}
```

Code Snippet 6.14: Function responsible for finding splitting of groups.

As a result of our interpolation model, we are able to apply a procedural animation to the droplet groups, while moving, merging and splitting. A static representation of an animation can be seen in Figure 6.21, and a video of the animations can be found in the footnotes of this page [1].

[1] Procedural Animation Video: https://youtu.be/g6BJ95sQk08

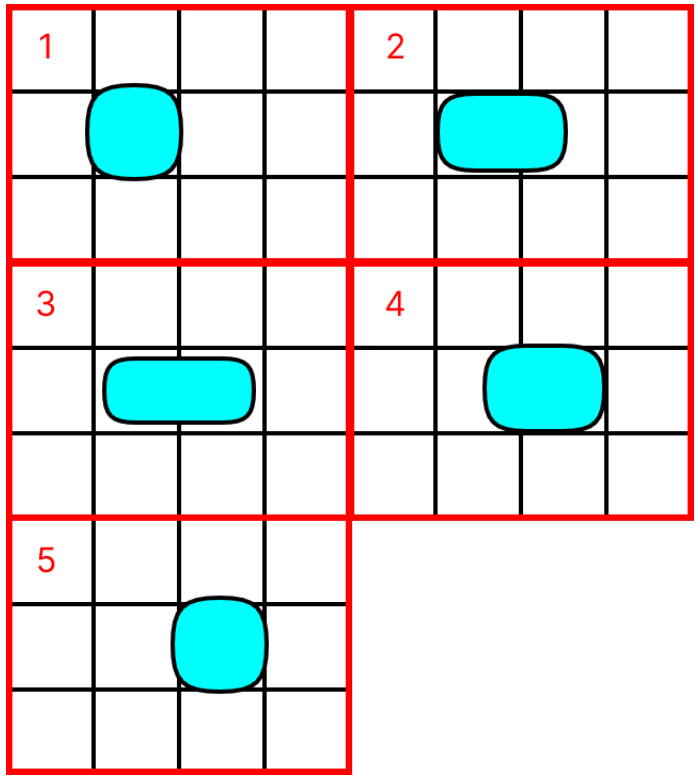

Figure 6.21: Static representation of the procedural animation of a droplet movement. 1) Initial position of the droplet. 2) Droplet in between the initial and elongated position. 3) Droplet in it's elongated position. 4) Droplet in between it's elongated and final position. 5) Droplet in it's final position.

View Summary

During this section, we presented how the Sketch Panel is drawn by the Sketch Panel Manager. This allowed for the visual representation of the simulation engine data on the GUI. In Figure 6.22 the GUI can be seen, where each design component that has been implemented is replaced by the actual GUI component.

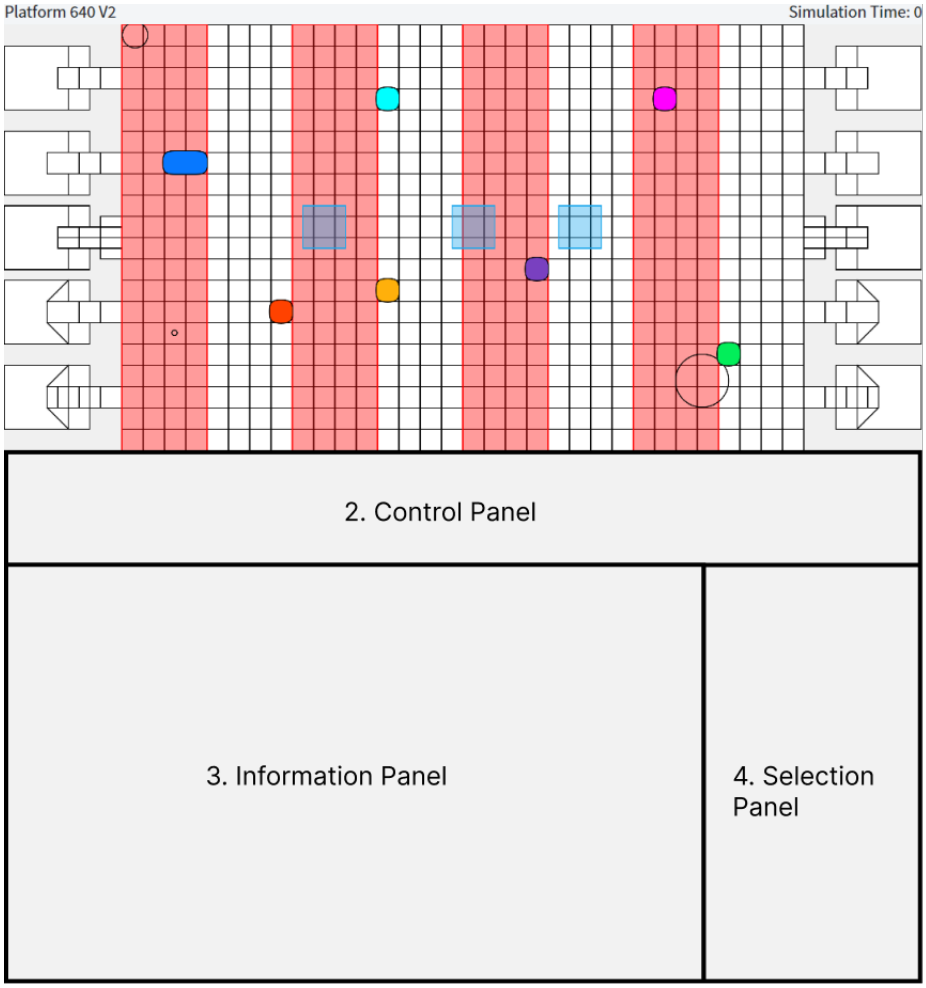

Figure 6.22: GUI design, with implemented Sketch Panel.

6.4 Control Panel

This section describes the design and implementation of the Control Panel and Control Panel manager, as seen in Figure 6.3.

Design

The Control Panels' overall design and placement in relation to the rest of the GUI can be seen in Figure 6.2, in Figure 6.3 the general placement and connection between the Control Panel Manager and Control Panel can be seen.

The Control Panel provides the user with necessary control over the simulation through the logic of the Control Panel manager; this includes stepping through the simulation, playing the simulation, going to a specific time in the simulation, restarting the simulation, and downloading data related to the simulation. The controls are connected to buttons, which are located on the Razor page called SimulatorPage; some on click callback functions are located within this page, providing direct calls to the simulation engine since a reference is present within the page. The functionalities located directly in the SimulaterPage are next step, play of the simulation, restart, and numeric input. There is no reason for these control functionalities to be handled in the Control Panel manager since this would require it to be held in JavaScript, making the calls more advanced for no reason and no improvement. The controls allow the user to navigate the simulation, moving back or forward to a specific point in time, or just running the simulation. For some components, specifically the numeric input and slider, we use custom components provided by Radzen[16].

Implementation

The Control Panel is structured by HTML, which is located within the SimulatorPage. A simplified HTML structure of the Control Panel can be seen in Code Snippet 6.15.

```
<div id="controlPanel">

    <div id="button_div">

        <div>
            <button>Next Step</button>
            <button>@PlayButtonText</button>
            <button>Restart</button>
            <button>Download</button>
        </div>

        <div>
            <span>TimeStep amount:</span>
            <RadzenNumeric></RadzenNumeric>
        </div>

    </div>
```

```
18
19      <div id="slider_div">
20          <RadzenSlider></RadzenSlider>
21          <button>Goto Time: @sliderValue</button>
22      </div>
23  </div>
```

Code Snippet 6.15: Simplified HTML of the Control Panel, function calls, classes, etc. is stripped from the snippet.

In Code Snippet 6.15, we can see that the Control Panel contains five buttons, a numeric input, and a slider. Together these interaction components provide the user with control over the simulation. For the numeric input and slider, we use the Radzen components, RadzenNumeric and RadzenSlider. These provide the exact functionalities needed. The RadzenNumeric component lets the user input a specified numeric value, in our case, a decimal value. The RadzenSlider provides a slider, which the user can manipulate by dragging. The slider has a minimum and maximum value, set to 0 and 100. The numeric input is used to determine the step time amount, in seconds, taken by the simulation engine when the next step button is clicked, where a time step amount of -1 indicates a step to the next action in the simulation engine action queue. The slider provides the user with a way of accessing a specific time point in the simulation. By dragging the slider to the exact point, the user is able to go to that exact time in the simulation. This can either be before or after the current simulation time.

6.4.1 Control Panel Manager

This section presents the Control Panel manager and its core functionality of allowing for the download of simulation data.

Design

The core functionality of the Control Panel manager is to provide the download functionality to the user. The download functionality allows the user to gather data about the simulator every given time step until a set time in the simulation is reached. This means that the user should be able to input the time step for which they would like to gather the data and an end time for which the data is gathered. To create a more advanced download menu for the user, a modal will be used, which essentially is a type of pop-up menu, that takes over the main focus of the website while active. The downloaded data will be in the file format

JSON and be a direct representation of the simulation engine's container data, which contains all information about the simulation. Code Snippet 6.16 displays the functions of the Control Panel manager related to downloading data.

```
/* Types */
/* The container relates to the simulation engine container data */
type data_for_download = container list

/* Functions */
function store_download_data (JSON) -> ()
function download_data: () -> ()
```

Code Snippet 6.16: Displays the download functionalities of the Control Panel manager, in a descriptive pseudo code.

We will present a short description of the functions in Code Snippet 6.16.

- **store_download_data(JSON)**
  Stores the data that should be downloaded into `data_for_download`, where it is stored until all data is gathered, and the `download_data` function is triggered.
- **download_data()**
  Downloads the data stored in `data_for_download` as a JSON file, providing all necessary data to the user.

With these functions, the Control Panel manager is able to provide a downloadable JSON file containing all data available. The user can then manipulate this data and create data representations from it.

6.4.2 Downloading Data

By the structure of the Control Panel HTML, a download button is present. By binding an on click function to this button, we can trigger a specific function when the button is pressed by the user. The on click function then triggers a modal to open, the modal is a pop-up menu containing input of two numerical values, one being the time step length that should be between each data point in the downloaded data, and the other being the end point time of the downloaded data, the modal can be seen in Figure 6.23.

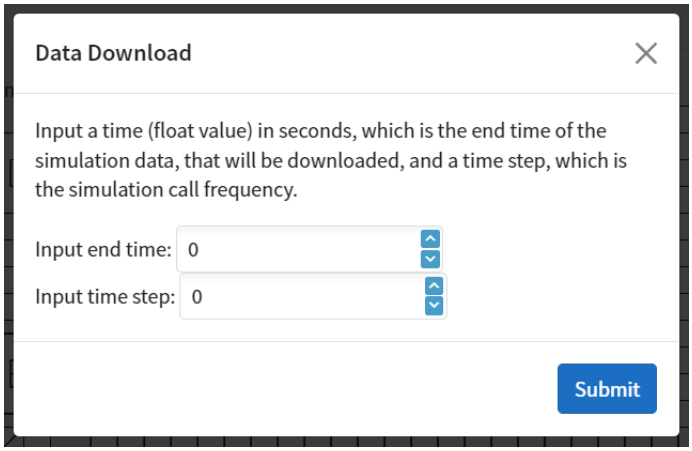

Figure 6.23: Download modal, displayed on top of all other GUI components, requiring an action to be taken before closing.

When the modal has been opened, two actions can be carried out, either the user can cancel the download by clicking the X in the top right corner or submit the download request by clicking submit. Submitting the request will trigger another function which will run the simulation engine, and at each time step inputted by the user, it will send the data through the GUI broker to the Control Panel manager, where it will be stored. When the simulation has reached the endpoint, inputted by the user, it will flag that it is done, and the `download_data` function in the Control Panel manager will be triggered. The `download_data` function attaches the data to a hidden HTML element, forcing a click on the element, thereby triggering a download of the data JSON file, and finishes by clearing the stored data. The `download_data` function can be seen in Code Snippet 6.17.

```
download_data() {
    let jsonData = JSON.stringify(this.data_for_download);

    /* Download the data by forcing a click on an anchor element. */
    var dataStr = "data:text/json;charset=utf-8," + encodeURIComponent(
                    jsonData);
    var dlAnchorElem = document.getElementById('downloadAnchorElem');
    dlAnchorElem.setAttribute("href", dataStr);
    dlAnchorElem.setAttribute("download", "data.json");
    dlAnchorElem.click();

    this.data_for_download = [];
}
```

Code Snippet 6.17: Displays the download functionalities of the Control Panel manager, in a descriptive pseudo code.

6.4.3 Example use of Data

By downloading the data, using the download feature, a program like Excel can be utilized to read the JSON file, and thereafter functionalities within excel can be used to manipulate the data, draw diagrams, etc. In Figure 6.24 an example of temperature over time for each droplet group can be seen.

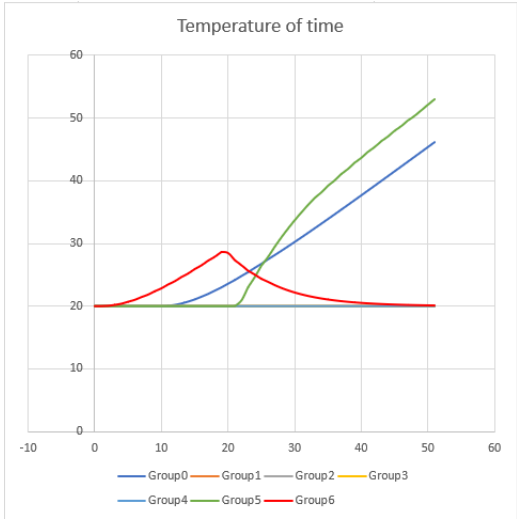

Figure 6.24: Temperature over time for each droplet group in the simulation.

From Figure 6.24 we can deduct that at time 50 seconds, group 5 has a temperature above $50°C$, group 4 has a temperature above $40°C$, and the rest of the groups has a temperature below $30°C$. This is a very simple example, but it gives insight into the actual temperatures during the simulation. For example, a protocol you write could have a group getting warmer than what was intended. Using this data, any such error can be found, and the protocol can then be rewritten.

View Summary

In this section, we described how the Control Panel and Control Panel manager has been designed and implemented. In Figure 6.29 the GUI can be seen, where each component that has been populated at this point is shown.

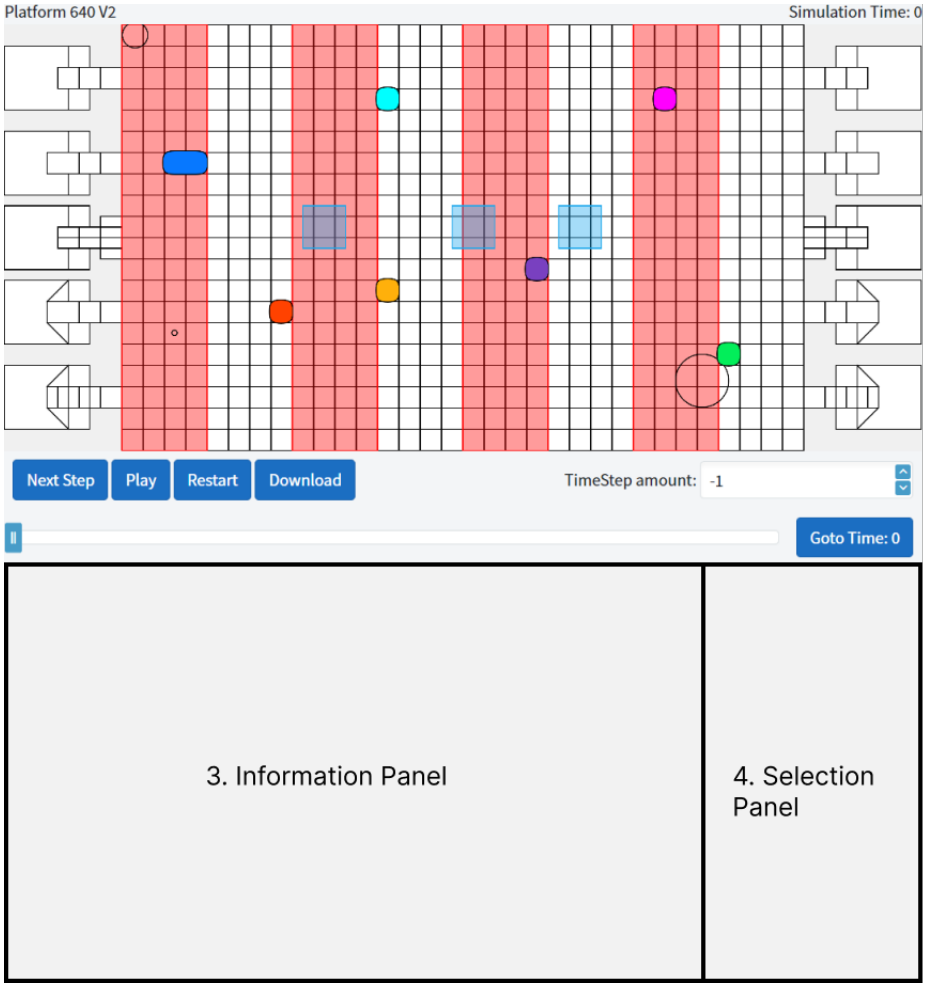

Figure 6.25: GUI design with implemented Sketch Panel and Control Panel.

6.5 Information Panel

This section describes the design and implementation of the Information Panel and Information Panel Manager, as seen in Figure 6.3.

Design

The Information Panels' overall design and placement in relation to the rest of the GUI can be seen in Figure 6.2, in Figure 6.3 the general placement and connection between the Information Panel Manager and Information Panel can be seen.

The main purpose of the Information Panel is to provide a way to make the simulation engine data visible to the user through the GUI. By clicking on different simulation elements in the Sketch Panel, their respective information is displayed on the GUI. The displayed information is filtered such that information only applicable to the simulation engine, which does not hold any important relevance to the user, is not shown. This is done to eliminate confusion and clutter and thereby provide a simplified version of the full data. If multiple elements have the same position in the Sketch Panel, a menu for selecting a specific element is provided, displaying every element in that exact location. The Information Panel Manager controls all business logic related to the information display, filtering, etc., while the Information Panel provides a designated place in the GUI where the selected data can be displayed. The Information Panel and Information Panel manager also provides a way for the user to alter simulation engine data, resulting in a change in the current running simulation. This feature is available through the Information Panel.

Implementation

The structure of the Information Panel is built by HTML, located with the rest of the frontend structure, in the Razor page called SimulatorPage. The HTML structure can be seen in Code Snippet 6.18.

```
<div id="informationPanel">
    <div id="informationTitle">
        <span>Information Panel</span>
    </div>

    <div id="information">
        <button id="edit_button" class="btn btn-primary"
            style="visibility: hidden">
                Edit
            </button>

        <div id="saveclose_button_div">
            <button id="save_button" class="btn btn-primary">
                Save
            </button>

```

```
            <button id="cancel_button" class="btn btn-primary">
                Cancel
            </button>
        </div>
        <div id="informationElements"></div>
    </div>
</div>
```

Code Snippet 6.18: HTML of the Information Panel.

In Code Snippet 6.18 we can see that the Information Panel contains different elements, such as buttons, which provides functionality that allows for the user to edit the data information displayed in the Information Panel, and thereafter either save the modifications or cancel them, returning to the previous state, with no modifications. The actual data will be listed in the div with id `informationElemetns`, here the data will be dynamically added by the Information Panel manager as input tags, meaning they are editable.

6.5.1 Information Panel Manger

This section describes the Information Panel manager, this includes the core logic of the Information Panel manager and the HTML DOM-manipulation [17] needed to display the information.

Design

The Information Panel manager contains key functionalities, which allows for the filtration of data, a menu for selecting elements when multiple elements are available, drawing the information onto the Information Panel, and dynamically updating information while the simulation is running.

```
/* Types */
type information        = (name * value) pair
type selected_element   = element
/* Functions */
function information_filter: string -> information list
function draw_multiple_selection: () -> ()
function draw_information: object -> ()
function dynamic_update: () -> ()
function on_edit: () -> ()
function on_save: () -> ()
function on_close: () -> ()
```

Code Snippet 6.19: Displays the core functions of the Information Panel manager, in a descriptive pseudo code.

In Code Snippet 6.19 the information is of the type key value pair, containing the name of the specific data and the value of the data.

A short description of each of the core functionalities will be presented.

- **information_filter(string)**
  The function information_filter takes a string argument related to the type of element for which the filtering should be carried out. It then filters the currently selected element through the specific type filter and returns a list containing the filtered information corresponding to the currently selected element.
- **draw_multiple_selection()**
  The function draw_multiple_selection is triggered when multiple elements are able to be selected from the same event. It then utilizes p5js to draw a menu on the Sketch Panel, allowing the user to choose between the different available elements.
- **draw_information()**
  The function draw_information draws the filtered information onto the Information Panel related to the currently selected element.
- **dynamic_update()**
  The function dynamic_update allows the data drawn onto the Information Panel by draw_information to update dynamically when the GUI receives new data from the simulation engine.
- **on_edit()**
  The function on_edit is triggered when the user presses the edit button, see Code Snippet 6.18, and allows for specific data to be edited through the Information Panel.
- **on_save()**
  The function on_save is triggered when the user presses the save button, see Code Snippet 6.18; it allows the user to save modifications that might have been made through the edit feature and send them to the simulation engine through the GUI broker.
- **on_cancel()**
  The function on_cancel is triggered when the user presses the cancel button, see Code Snippet 6.18, allowing the user to revert any changes that might have been made using the edit feature.

These functions provide the Information Panel manager with the ability to draw information, change information, and filter information onto the Information Panel.

6.5.2 Displaying Information

When the user has clicked on an element in the Sketch Panel, we want to draw the corresponding information, but with the ability to selectively chose what information is displayed. Therefore the information is passed through the `information_filter` function. The `information_filter` takes a type, corresponding to the type of the element that has been selected by the user, such as an electrode, a droplet, an actuator, a sensor, etc. The Information Panel manager stores a list of attributes for each element, which are the values that are allowed to be displayed on the Information Panel.

To illustrate the process of filtering and displaying information, an example, where an electrode has been chosen by the user can be considered. The Information Panel manager receives a call to the `information_filter` through a callback from a mouse click event, which is set off in the Sketch Panel manager. The `information_filter` receives the type `"Electrode"` as it's argument, compares this in a switch statement, see Code Snippet 6.20.

```
information_filter(type) {
    let returnVal;

    switch(type) {
        ...
        case ("Electrode"):
            let electrode = arguments[1];

            /* Gather the valid attributes */
            returnVal = this.display_info[type];

            /* Populate valid attributes from the electrode */
            for (let key in returnVal) {
                returnVal[key] = electrode[key];
            }

            returnVal.Type = type;

            /* Return an object with populated valid attributes */
            return returnVal;

            break;
        ...
    }
}
```

Code Snippet 6.20: Snippet of the switch statement responsible for handling information filtering.

Code Snippet 6.21 presents the part of the list of valid attributes related to the electrode element. The list also contains an array of attributes which are able

to be edited by the user, the list of editable values will be used later in this section when we implement the edit functionality.

```
display_info {
    ...,
    Electrode: {
        name: "",
        ID: "",
        status: 0,
        positionX: 0,
        positionY: 0,
        subscriptions: []
    },
    Electrode_editable: ["status"],
    ...
}
```

Code Snippet 6.21: Attribute filtering list related to the electrode element.

At this point, the object returned from `information_filter` contains all the information that should be displayed on the Information Panel. To display the object's attributes onto the Information Panel, it is passed into the `draw_information` function as an argument. The `draw_information` function gets a reference to the informationElements div, then iterates through the object passed in and creates an input tag, assigned with a read only flag, for every attribute in the object. If it is not flagged for read only, the input tag allows for its value to be edited by the user as a text field. The input tags are then appended into the informationElements div and are then visible in the Information Panel. Figure 6.26 shows the Information Panel after the electrode with ID 0 has been selected by the user.

Figure 6.26: Information Panel displaying electrode with ID 1.

The information displayed in Figure 6.26 is the filtered information. As a reference, Code Snippet 6.22 contains all information available about the electrode.

```
/* Electrode attributes */
ID: 1
corners: []
driverID: 0
electrodeID: 360
name: "arrel1"
neighbours: [2, 33]
positionX: 110
positionY: 0
shape: 0
sizeX: 20
sizeY: 20
status: 0
subscriptions: []
```

Code Snippet 6.22: All attributes of the electrode with ID 1.

6.5.3 Editing Information

Next, the issue of editing information through the Information Panel and then updating it in the simulation engine will be tackled. As seen in Section 6.5, the structure of the Information Panel contains three different buttons, an edit button, a save button, and a cancel button. Each of these buttons triggers a corresponding function in the Information Panel manager; these functions are responsible for handling the different events. The save and cancel button is hidden by default, and only the edit button is visible. To achieve the above functionality, the edit button and the corresponding function `on_edit`, will be presented and implemented.

When the edit button is clicked by the user, an event is fired, resulting in the `on_edit` function being called. The first action of the function is then to hide the edit button and make the save and cancel button visible. Next, `on_edit` fetches the allowed editable values for the selected element type, the editable values for an electrode can be seen in Code Snippet 6.21. The input tag corresponding to the editable attribute is then found, and the read only flag is removed. This allows the user to alter the value of the input tag, as seen in Figure 6.27.

Figure 6.27: Information Panel displaying electrode with ID 1 after the edit button has been clicked.

When an attribute has been edited, the user has two options, cancel or save the edit. Cancelling will trigger the `on_cancel` function, which resets the attributes to their original values, hides the save and cancel buttons and makes the edit button visible again. Saving the edit will trigger a function in the GUI broker, function `update_simulator_container`, which takes an element type and a JSON object as arguments. The GUI broker transfers the data to the simulation engine, where the data will be deserialized from JSON to the respective element, the corresponding element in the data of the simulation engine is then found based on the ID of the element. The values of the simulation engine element are then updated to the ones of the exchanged object.

6.5.4 Multiple Selection

When multiple elements overlap on the Sketch Panel, there needs to be a way to select between the elements, otherwise, the element drawn on top will take priority and get selected. To allow flexibility for the user, a menu to select between multiple elements is implemented. By double clicking on the Sketch Panel, if more than one element is located in the clicked position, a menu is drawn on top of the Sketch Panel. The menu is drawn by p5js and is built by a rectangle with rounded edges and text inside; the menu can be seen in Figure 6.28. When the menu is activated, N actions can be taken, where N is the number of elements positioned at the same location. By clicking a keyboard key corresponding to the one shown in the menu, the user will be able to select

between the different elements.

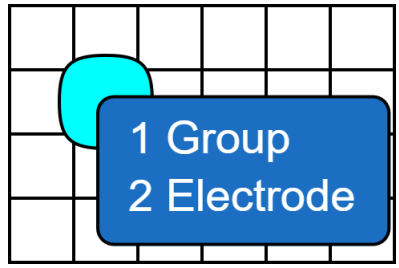

Figure 6.28: Droplet group and electrode located at the same position, the action of double clicking trigger the multiple select menu where two actions are possible 1) the droplet group is selected, 2) the electrode is selected.

View Summary

This section describes how the Information Panel and Information Panel manager has been designed and implemented and how the Information Panel is populated. In Figure 6.29 the GUI can be seen, where each component that has been populated at this point is shown.

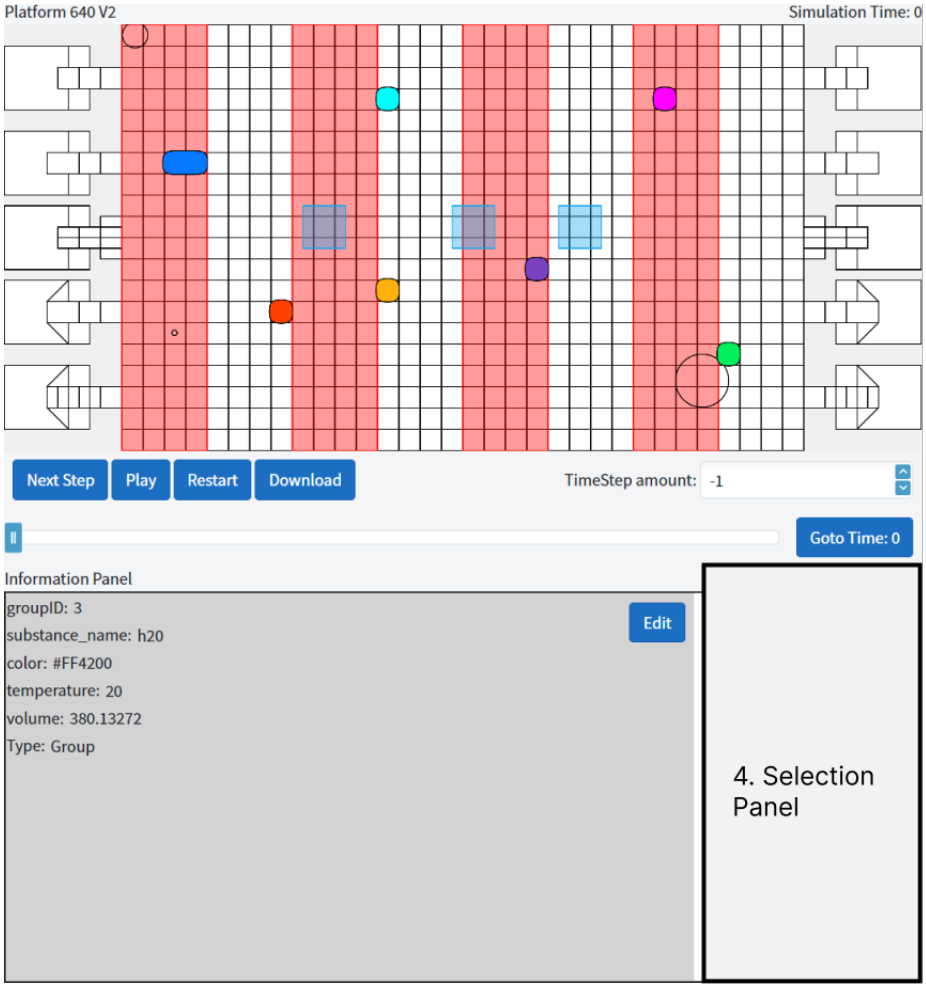

Figure 6.29: GUI design, with implemented Sketch Panel, Control Panel, and Information Panel.

6.6 Selection Panel

This section presents the design and implementation of the Selection Panel and the Selection Panel manager, which can be seen in Figure 6.3

Design

The Selection Panels' overall design and placement in relation to the rest of the GUI can be seen in Figure 6.2. In Figure 6.3 the general placement and connection between the Selection Panel Manager and the Selection Panel can be seen.

The Selection Panel provides a way for the user to select which components and properties of the simulation should be applied and shown on the Sketch Panel. This allows the user to selectively add or remove visual components such as actuators, sensors, etc., or choose applied properties, such as procedural animations, real-time execution, etc. The components and properties are presented as elements in a list located in the Selection Panel, where each element is toggleable. The list of toggleable elements is not a complete list but is modular, meaning new selection elements can be added to offer the possibility of expansion on components and properties.

Implementation

The Selection Panel is structured by HTML, which is located and combined with the rest of the frontend structure in the Razor page called SimulatorPage. Code Snippet 6.23 displays the structure of the Selection Panel.

```
<div id="selectionPanel">
    <span>Selection Panel</span>
    <form name="layer">
        /* elements are inserted here */
    </form>
</div>
```

Code Snippet 6.23: Selection Panel structure in HTML.

6.6.1 Selection Panel Manager

This section will present the Selection Panel manager, responsible for handling all logic related to the Selection Panel.

Design

The Selection Panel manager stores the list of possible selections and provides the utility to add togglable elements to the Selection Panel. In Code Snippet 6.24 the core functionalities of the Selection Panel manager can be seen.

```
1
2  /* Attributes contained by all elements */
3  type name = string
4  type value = string
5  type id = string
6  type text = string
7  type element = object
8  type checkbox = object
9  type checked = boolean
10 type layer: = object // (optional)
11
12 type element = name * value * id * text * element * checkbox * checked
13                * layer
14
15 type selection_list = element list
16
17 /* Functions of Selection Panel manager */
18 function initialize_selection: () -> ()
19 function draw_layers: () -> ()
```

Code Snippet 6.24: Displays the core functions of the Selection Panel manger, in a descriptive pseudo code.

A short description of the functions in Code Snippet 6.24 are presented.

- **initialize_selection()**
  The function `initialize_selection` goes through the list of selections and creates an entry in the Selection Panel corresponding to the element in the selection list.
- **draw_layers()**
  The function `draw_layers()` draws the layer of all selections that contains the optional attribute layer.

The checkbox attribute contains a reference to the actual checkbox element, which is added to the Selection Panel through `initialize_selection`; the attribute checked allows for the selection to initially be visible or hidden in the Sketch Panel. Elements and properties are only applied to the visual representation if the checkbox is checked, thereby providing a way to enable or disable attributes from the visual representation of the simulation.

6.6.2 Managing the Selection

To allow the user to select which components and properties that should be applied to the visual representation of the simulation, the Selection Panel manager needs to supply an interaction point for the user. When the simulation is first loaded, the function `initialize_selection` is triggered, this function uses HTML DOM manipulation to add an input tag for each element in the selection_list. The `initialize_selection` function can be seen in Code Snippet 6.25

```
initialize_selection() {
    /* Iterate through the selection_list */
    for (let layer in this.selection_list) {

        /* Create HTML elements */
        var div = document.createElement('div');
        div.classList.add("form-check");

        /* Construct the Checkbox */
        div.innerHTML =
        `<input type='checkbox' name='${this.layers[layer].name}'
        value='${this.layers[layer].value}'
        id='${this.layers[layer].id}'
        class='form-check-input'/>
        <label for='${this.layers[layer].name}'
        class='form-check-label'>${this.layers[layer].text}</label>`;

        /* Update entries in selection panel manager */
        this.layers[layer].element = div;
        this.layers[layer].checkbox = div.querySelector('input');
        this.layers[layer].checkbox.checked = this.layers[layer]
                                                    .checked;

        /* Add element to the Selection Panel */
        document.querySelector("#layerPanel").querySelector('form')
                                                    .append(div);
    }
}
```

Code Snippet 6.25: Code of the function initialize_selection, used to initialize the HTML elements of the Selection Panel.

An entry of the selection list can be seen in Code Snippet 6.26

```
selection_list = {
    ...,
    draw_actuators: {
        name: "draw_actuators",
        value: "draw_actuators",
        id: "draw_actuators",
        text: "Draw Actuators", // Shown in the Selection Panel
```

```
        element: "insert",
        checkbox: "insert",      // Toggleable
        checked: true,
        layer: "insert"          // Optional
    },
    ...
}
```

Code Snippet 6.26: Displays the selection of drawing actuators, located in the selection list.

To add another selection to the Selection Panel, an entry into the selection list needs to be made; this entry should adhere to the structure described in Code Snippet 6.24 and 6.26.

After running `initialize_selection`, the Selection Panel is populated with the toggleable selection options, as can be seen in Figure 6.30.

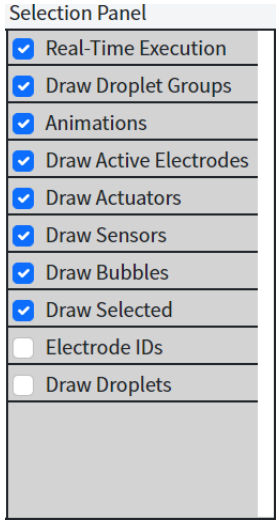

Figure 6.30: Selection Panel populated by the Selection Panel manager, with toggleables.

View Summary

During this section, we presented the Selection Panel and the Selection Panel manager and how they allowed for the selection of which components and properties should be applied in the visual representation of the simulation.

Every component of the GUI has been populated at this point, meaning that the final product of the GUI, as shown in Figure 6.1, has been achieved. The final product can also be seen in Figure 6.31.

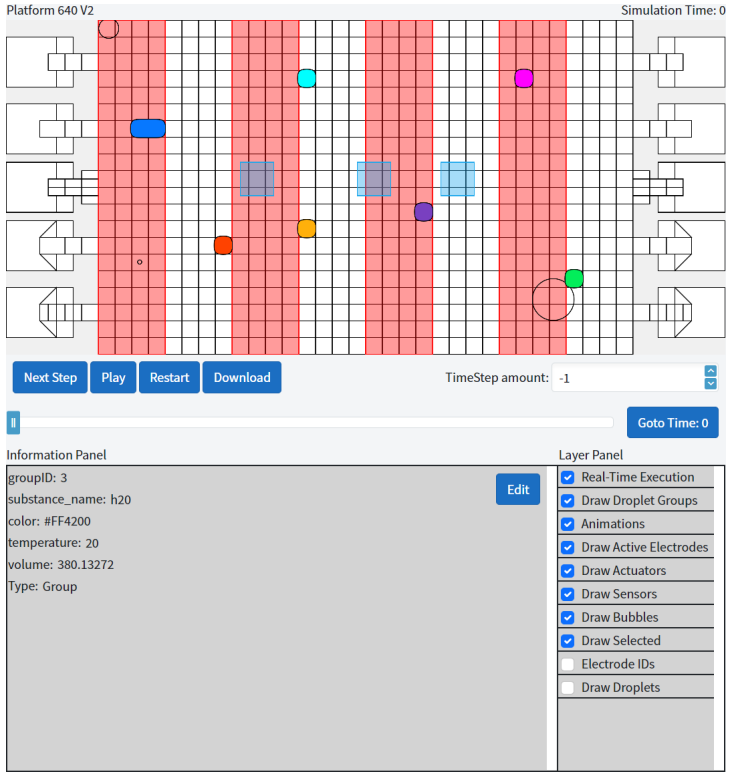

Figure 6.31: GUI with the, Sketch Panel, Control Panel, Information Panel, and Selection Panel populated.

6.7 GUI Chapter Summary

In this chapter, we described the design and implementation of the frontend and backend parts of the GUI. By dividing the GUI into components, we were able to build the GUI component by component through a set of sections, each section adding a new component to the GUI, resulting in the complete GUI. Through those sections, we introduced and added different functionalities, including a way to visualize the simulation engine, control the simulation, view and edit the data of different elements in the simulation, and select which components/prop-

erties should be applied to the visual representation of the simulation. These functionalities provide the user with the ability to modify the simulation and configure the visual representation to their liking, selecting which properties that are of importance to them. By this, the GUI both provides a visual representation of the DMF platform, simulated through the simulation engine and models, and also an interaction point, where the user can interact directly with the data for which the simulation engine runs.

Chapter 7

Simulation Interchangeability

To highlight the modularity of the Simulation Engine, this chapter introduces the user to the necessary knowledge for modifying or adding models of an existing component and adding a new component along with its models to the simulator. A new component can, for example, be an additional actuator or sensor. Section 7.1 explains how to change or add a model to a droplet. The remaining sections of this chapter go through all the steps in an example of implementing a new actuator called a vibrating micro-shaker. This chapter is very technical and includes a lot of code; this is necessary for explaining the details of adding a component.

7.1 Adding or Changing a Model of an Existing Component

When making the simulator, the changeability of models was important. Changeability means that future development, including changes to models or the addition of new models, should be simple. In this section, we will use the droplet models as a foundation for explaining how to change and add a model to an existing component.

Changing a Model

To change a model, one can either replace the class of a model, change a method in the class, or do other edits to the existing class. These could be smaller or larger changes; the only thing of importance is always to return the subscribers at the end of the model. The subscriber array should always include the caller [1], except if the caller is removed. The array can also contain additional subscribers.

Adding a Model

Adding a completely new model is slightly more complex. In this case, the user has to implement the new model from scratch, and it must adhere to the rule of returning subscribers. Once the new model is ready and put in its own class, a name for the model must be added to the array of the droplets models. It also has to be added in the switch statement as a new switch case; the switch statement itself can be seen in Code Snippet 7.1.

```
1  private static ArrayList executeModel(Container container, Droplets
                       caller, String model)
2          {
3              switch (model)
4              {
5                  case "split":
6                      return Models.DropletModels.dropletSplit(container,
                           caller); ;
7                  case "merge":
8                      return Models.DropletMerge.dropletMerge(container,
                           caller);
9                  ...
10
11             }
12             return null;
13         }
```

Code Snippet 7.1: Model switch statement.

7.2 Initialization and Datatype

The following sections follow the example of adding a new component and its models to the simulation framework. The new component we use as an example is called a vibrating micro-shaker. The micro-shaker is a type of actuator. It

[1] A caller is described in the Appendix entry “Caller” A.2.

requires the addition of a new entry to the JSON file, which contains the fields needed for its functionality; these are vibration frequency, power status, name, ID, type, position and size fields. Furthermore, it requires new models to be implemented, both to model the behaviour of the micro-shaker and to model the behaviour of the components which the micro-shaker affects.

7.2.1 Adding a Component to the JSON File

The first step of implementing a micro-shaker is to add it to the JSON configuration file. Since we want to implement a micro-shaker, it has to be added to the actuator section of the JSON file.
The type of the actuator should be specified in the name field, and all the required fields of an actuator must be filled. The required fields can be deduced by looking at the fields of the actuator class while not considering fields from classes that extend the actuator.
The user also needs to specify any additional fields that they want to get the functionality of the actuator to work. In the example of adding a micro-shaker, the user would have to add a field for vibration frequency and a field for the power status of the micro-shaker.
An example of adding a micro-shaker to the JSON file can be seen here.

```
"actuators": [
    {
      "valueActualTemperature": 20,
      "valueDesiredTemperature": 20,
      "valuePowerStatus": 0,
      "name": "heatactuator1",
      "ID": 727,
      "actuatorID": 1,
      "type": "heater",
      "positionX": 110,
      "positionY": 0,
      "sizeX": 80,
      "sizeY": 400
    },
    {
      "valueActualTemperature": 20,
      "valueDesiredTemperature": 20,
      "valuePowerStatus": 0,
      "name": "heatactuator2",
      "ID": 728,
      "actuatorID": 2,
      "type": "heater",
      "positionX": 270,
      "positionY": 0,
      "sizeX": 80,
      "sizeY": 400
    },
```

```
28      // The new actuator is added below.
29      {
30        "valueVibrationFrequency": 0,
31        "valuePowerStatus": 0,
32        "name": "micro-shakeractuator1",
33        "ID": 729,
34        "actuatorID": 3,
35        "type": "micro-shaker",
36        "positionX": 100,
37        "positionY": 100,
38        "sizeX": 40,
39        "sizeY": 160
40      }
41  ]
```

Code Snippet 7.2: Example of adding a micro-shaker actuator to the JSON file.

7.2.2 Adding a Component to the Container

The new actuator will be retrieved from the JSON file when initializing the Simulation Engine. The container needs a way to store the new component; therefore, we have to add an extra class and add the new get and set function, such as ”valueVibrationFrequency”, to the already existing actuator class. If we were implementing an additional sensor, the functions would have to be added to the sensor class. This can be done as in Code Snippet 7.3.

```
1   ...
2       public int positionX { get; set; }
3       public int positionY { get; set; }
4       public int sizeX { get; set; }
5       public int sizeY { get; set; }
6
7       // Add the new get and set function for the micro-shaker
8
9       public float valueVibrationFrequency { get; set; }
10      public float valuePowerStatus { get; set; }
11  ...
```

Code Snippet 7.3: Adding get/set method to actuator class.

Now we can construct the new class “micro-shaker” that extends the actuator class. Constructing the “micro-shaker” class can be done the same way the heater is implemented and can be seen here.

```
namespace MicrofluidSimulator.SimulatorCode.DataTypes
{
    public class MicroShaker: Actuators
    {
        public MicroShaker(string name, int ID, int actuatorID, string
                    type, int positionX, int positionY, int sizeX,
                    int sizeY, float valueVibrationFrequency, float
                    valuePowerStatus)
            :base(name, ID, actuatorID, type, positionX, positionY,
                    sizeX, sizeY)
        {
            this.valueVibrationFrequency = valueVibrationFrequency;
            this.valuePowerStatus = valuePowerStatus;


        }
        /* Any additional functions needed for the functionality of the
                     micro-shaker can also be added here, for example
                     */
        public float GetVibrationFrequency()
        {
            return valueVibrationFrequency;
        }

        public float GetCurrentPower()
        {
            return valuePowerStatus;
        }

    ...
    }
}
```

Code Snippet 7.4: Adding the micro-shaker class.

The container now has all the functionality for containing a micro-shaker. The next step is to initialize the component.

7.2.3 Initializing a Component

Initializing a new component can be done in the same way as all the other components; this is elaborated on in Section 5.4.1.
There already exists an initialize function for the actuators in the initialize class. The function is called **initializeActuators**. It switches on the actuator's type and initializes the specific fields accordingly.
To initialize the micro-shaker, the user must add a case to the switch statement. The switch case must adhere to the "type" given in the JSON file. Any

additional fields can also be initialized here; keep in mind that they need to be included in the micro-shaker class.
After adding the switch case, the micro-shaker will be stored in the same actuator array as the remaining actuators.
An example can be seen here.

```
private DataTypes.Actuators[] initializeActuators(Actuators[] actuators
                    )
        {

            DataTypes.Actuators[] actuatorsInitial = new DataTypes.
                    Actuators[actuators.Length];

            for(int i = 0; i < actuators.Length; i++)
            {
                switch (actuators[i].type)
                {
                    case "heater":
                        actuatorsInitial[i]= (new Heater(actuators[i].
                      name, actuators[i].ID, actuators[i].actuatorID,
                      actuators[i].type, actuators[i].positionX,
                            actuators[i].positionY, actuators[i].sizeX,
                       actuators[i].sizeY, actuators[i].
                      valueActualTemperature, actuators[i].
                      valueDesiredTemperature,
                            actuators[i].valuePowerStatus));

                        break;
                    // A new case for the micro-shaker is added
                    case "microShaker":
                        actuatorsInitial[i]= (new MicroShaker(actuators
                      [i].name, actuators[i].ID, actuators[i].
                      actuatorID, actuators[i].type, actuators[i].
                      positionX,
                            actuators[i].positionY, actuators[i].sizeX,
                       actuators[i].sizeY, actuators[i].
                      valueVibrationFrequency,
                            actuators[i].valuePowerStatus));

                        /* Adding additional micro-shaker specific
                      fields is easily done as follows */
                        actuatorsInitial[i].specificField = someValue;
                        break;
                }
            }

            return actuatorsInitial;

        }
```

Code Snippet 7.5: Initializing the micro-shaker.

The container now holds the new actuator, and the models and functionality of

the micro-shaker can be implemented.

7.3 Adding New Models

7.3.1 Adding Component Specific Models

The first step of adding new models is to create a new class for the models in the folder "MicrofluidSimulator/SimulatorCode/Models".
Let us call this class "VibratorActuatorModels". In the class "VibratorActuatorModels", the user can add all the models necessary for achieving the wanted functionality of the vibrator.
An example can be a dynamic model such as "setVibrationFrequency". The user can specify the desired vibration frequency, and the power status of the vibrator will be changed accordingly.

```
1 public static void microShakerVibrationFrequencyChange(Container
                      container, MicroShaker michroShaker)
2         {
3
4             microShaker.valueVibrationFrequency = /* some calculation
                      possibly dependent on container.timeStep and
                      microShaker.valuePowerStatus */
5
6         }
```

Code Snippet 7.6: Adding the vibration frequency change model.

The models for the actuators are run at every time step. To run the newly implemented model, the user has to add a new switch case in the simulator function **executeActuatorModel**. The switch statement is somewhat similar to the one mentioned in Section 7.2.3. Adding the case is easily done as follows:

```
1 private static void executeActuatorModel(Container container, Actuators
                       actuator)
2         {
3
4             switch (actuator.type)
5             {
6                 case "heater":
7
8
9                     HeaterActuatorModels.heaterTemperatureChange(
                        container, (Heater) actuator);
10                    break;
11                /* The new case, which adheres to the type of the
                        actuator, is added below */
```

```
12                 case "microShaker":
13
14
15                     MicroShakerActuatorModels.
                         microShakerVibrationFrequencyChange(container, (
                         MicroShaker) actuator);
16                     break;
17             }
18
19         }
```

Code Snippet 7.7: Adding vibration frequency change model to switch statement.

The model will now be run, and the vibration frequency is being changed based on some user-defined calculations.

7.3.2 Adding Component Depending Models

The user possibly wants the newly implemented vibration frequency of the actuator to act on other components of the simulator, such as droplets.
To implement this, the user has to add a new droplet model. This is done by adding a new class for the models in the "MicrofluidSimulator/SimulatorCode/-Models" folder.
Let's call this class "DropletVibrationModels". The user can implement a simple model for the behaviour of a droplet when a vibration is acting on it.
Any droplet model must always return an arraylist of the droplet subscribers as mentioned in Section 5.3.1. Following the implementation of the model **dropletTemperatureChange** we can implement a new model **dropletVibrationChange**. The new function is added in the "DropletVibrationModels" class.

```
1  public static ArrayList dropletVibrationTemperatureChange(Container
                       container, Droplets caller)
2          {
3              ArrayList subscribers = new ArrayList
4              {
5                  caller.ID
6              };
7
8
9              MicroShaker microShaker = (MicroShaker)
                       HelpfullRetreiveFunctions.getActuatorOnDroplet(
                       container, caller, "microShaker");
10
11             if(microShaker != null)
12             {
```

```
13                  /* In this example, the temperature of the droplet is
                       changed based on the vibration frequency of the
                       micro-shaker */
14
15
16                  caller.temperature += /* some calculation based on
                       microShaker.GetVibrationFrequency, container.
                       timeStep and caller.mass */ ;
17                  return subscribers;
18
19              }
20              return subscribers;
21
22          }
```

Code Snippet 7.8: Adding the droplet vibration model.

The newly created droplet model has to be added to the switch case in the simulator that executes the droplet models. Adding the model to the switch case in the simulator in the function **executeModel** can be done as in Code Snippet 7.9.

```
1  private static ArrayList executeModel(Container container, Droplets
                      caller, String model)
2          {
3              switch (model)
4              {
5                  case "split":
6                      return Models.DropletModels.dropletSplit(container,
                           caller); ;
7                  ...
8                  case "vibrateDroplet":
9                      return Models.DropletVibrationModels.
                         dropletVibrationTemperatureChange(container,
                         caller);
10             }
11             return null;
12         }
```

Code Snippet 7.9: Adding the droplet vibration model to the switch case.

The model has to be added to the model order in the droplet class found in the folder ”MicrofluidSimulator/SimulatorCode/DataTypes”. It is important to add it in the correct position in the array. The model's position in the array decides when the model should be executed relative to the other models in the array, this is further explained in Section 5.1. Since this is merely an example; the model will be added to the end of the array as seen in Code Snippet 7.10. The model is now being run if a droplet is on top of a vibrator. The last step is to visualize the vibrator in the graphical user interface.

```
...

    public Droplets(string name, int ID, string substance_name, int
                    positionX, int positionY, int sizeX, int sizeY,
                    string color, float temperature, float volume,
                    int electrodeID, int group, double
                    accumulatingBubbleEscapeVolume)
            {
                ...

                nextModel = 0;
                beginOfTimeSensitiveModels = 3;

                /* The name used in the modelOrder array has to adhere
                       to the name used in the switch case for executing
                        the droplet models */
                modelOrder = new string[] {"split", "merge", "split", "
                       color", "temperature", "makeBubble", "
                       vibrateDroplet"};
                ...
            }
...
```

Code Snippet 7.10: Adding the droplet vibration model to the model order array in the droplet class.

7.4 Visualizing a Component

This section describes the process of visualizing a new component, using the example of the vibrating micro-shaker.

This will be done through four steps;

1. Designing the component.
2. Drawing the component.
3. Conditional drawing of the component.
4. Selecting information about the component.

Designing the Component

To visualize a new component, the first step is to design a shape that is distinguishable from the other components. Firstly we can use a colour that is not yet

used by the current components, such as green, and add a circular shape to the middle, representing the micro-shaker. The design of the vibrating micro-shaker can be seen in Figure 7.1.

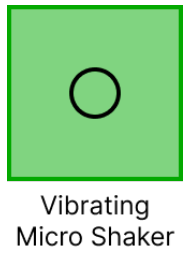

Figure 7.1: A possible design for a vibrating micro-shaker component.

This design is distinguishable, meaning it would be easy to tell apart from the rest of the simulator components, this is important in order to avoid possible confusion.

Drawing the Component

The next step in visualizing the component is to convert the design into code using the p5js framework. To do this, we first boil down the design into the basic shapes. The basis of the component consists of a green square with a green border. Inside the square, there is a circle without any background and a black border. Code Snippet 7.11 displays the code, which draws the design of the vibrating micro-shaker.

```
function draw_vibrating_micro_shaker() {
    let micro_shakers = gui_broker.board.micro_shakers;

    micro_shakers.forEach((micro_shaker) => {
        let color = p.color("#03A900");
        color.setAlpha(100);
        p.fill(color);
        p.strokeWeight(4);
        p.stroke("#03A900");

        p.rect(micro_shaker.x, micro_shaker.y, micro_shaker.sizeX,
                                               micro_shaker.sizeY);

        p.noFill();
        p.stroke("#000000");
        /*
        * Position in center of square.
        * Static size of 5 pixels.
        */
        p.ellipse(micro_shaker.x + micro_shaker.x/2,
                  micro_shaker.y + micro_shaker.y/2, 5);
    )}
}
```

Code Snippet 7.11: Drawing the vibrating micro-shaker component.

To draw the component onto the Sketch Panel, the draw function has to be added to the p5js draw loop within the Sketch Panel manager 6.3.3.

Conditional Drawing of the Component

To add flexibility to the user, in the same manner as the other components, the drawing of the component should be toggleable. To make the drawing of the component toggleable, it has to be added to the Selection Panel manager 6.6.1. This is done by adding it to the selection list, as seen in Code Snippet 7.12.

```
selection_list = {
    ...,
    draw_micro_shaker: {
        name: "draw_micro_shaker",
        value: "draw_micro_shaker",
        id: "draw_micro_shaker",
        text: "Draw Micro-Shaker",
        element: "insert",
        checkbox: "insert",
        checked: true,
        layer: "insert"
    },
    ...
}
```

Code Snippet 7.12: Entry of the micro-shaker into the Selection Panel manager.

Selecting Information about the Component

The last step for the component is to make it selectable and display information within the Information Panel when selected, for which the Information Panel manager from Section 6.5.1 is utilized.

First, an entry in the `display_info` is needed, the entry determines which values should be drawn onto the Information Panel, as well as which values should be editable by the user.

```
display_info: {
    ...
    MicroShaker: {
        name: "",
        ID: 0,
        actuatorID: 0,
        type: "",
        valueVibrationFrequency: 0,
        valuePowerStatus: 0
    },
    MicroShaker_editable: [] // Nothing can be edited
    ...
}
```

Code Snippet 7.13: Information added to the display_info in the Information Panel manager.

The component now has to be added to the `information_filter` which is used to filter the actual information, to the desired information stated in Code Snippet 7.13. The entry in the `information_filter` can be seen in Code Snippet 7.14.

```
information_filter: function(type) {
    let returnVal;
    this.selected_element_type = type;

    switch(type) {
        ...
        case("MicroShaker"):
            // Get the micro-shaker object
            let microShaker = arguments[1];

            // Get the desired information
            returnVal = this.display_info[type];

            // Let returnVal contain all desired information
            for (let key in returnVal) {
                returnVal[key] = microShaker[key];
            }

            returnVal.Type = type;

            // Return
            return returnVal;
        ...
    }
}
```

Code Snippet 7.14: Micro-shaker entry into the information_filter function.

To show any information about the micro-shaker component, it has to be clickable. An entry in the Sketch Panel onClick event must be made to achieve this. The Sketch Panel onClick event is located in the Sketch Panel manager, within the function `onMouseClicked`. Since this component is of the actuator type, we have to make modifications to the actuator onClick handler.

```
function onMouseClicked() {
    ...
    // Handle click on actuator
    if (selection_manager.layers.draw_actuators.checkbox.checked) {
        ...
        // If cursor is on an actuator
        if (polygon_contains(vertexes, p.mouseX, p.mouseY)) {
            information_panel_manager.selected_element = actuator;

            // If the actuator is a micro-shaker
            if(actuator.type == "MicroShaker") {
                information_panel_manager.information_element =
                information_panel_manager.information_filter(
                                    "MicroShaker", actuator);

                information_panel_manager.draw_information(
                information_panel_manager.information_filter(
                                   "MicroShaker", actuator));

            // Default
            } else {
                information_panel_manager.information_element =
                information_panel_manager.information_filter(
                                      "Actuator", actuator);

                information_panel_manager.draw_information(
                information_panel_manager.information_filter(
                                    "Actuator", actuator));
        }
        ...
        }
    }
    ...
}
```

Code Snippet 7.15: Micro-shaker entry added to the onClick handler for actuators.

Code Snippet 7.15 shows the onClick handler for the actuators, with the added handler for the vibrating micro-shaker. The component is now selectable by clicking on it in the Sketch Panel; however, it is not yet present in the multiple select menu. To make the micro-shaker selectable in the multiple select menu, we have to add it to the onDoubleClick handler, located in the Sketch Panel manager. The addition of the micro-shaker to the onDoubleClick handler can be seen in Code Snippet 7.16.

```
function onMouseDoubleClicked() {
    ...
    // Handle click on actuator
    if (selection_manager.layers.draw_actuators.checkbox.checked) {
        for (let i in gui_broker.board.actuators) {
            ...

            // If the cursor is on an actuator
            if (polygon_contains(vertexes, p.mouseX, p.mouseY)) {
                if(actuator.type == "micro-shaker") {
                    list_of_elements["Micro Shaker"] = actuator;

                // Default
                } else {
                    list_of_elements["Actuator"] = actuator;
                }
            }

            ...
        }
    }
    ...
}
```

Code Snippet 7.16: Micro-shaker entry added to the onDoubleClick handler for actuators.

Chapter 8

Evaluation and Testing

This chapter presents an evaluation of the simulation framework and the tests that have been carried out, to ensure the functionality of the simulation framework.

8.1 Browsers and Operating Systems

The simulation framework has been compiled and run on different browsers and operating systems. The following browsers support running the framework; Microsoft Edge, Firefox, Google Chrome, and Opera. It was run on the following operating systems; Windows 10 and macOS Big Surr.

8.2 Unit Testing

Unit testing was done using the unit-testing framework NUnit [18] for C#. The framework allows for thorough testing of the code. The test prioritization is further discussed in Section 9.1. We constructed tests for five different functionalities. These include:

- Neighbour finder
- Subscription initialization
- Droplet group initialization
- Droplet movement
- Droplet split and merge

The tests are simple non-parameterized tests using the "[Test]" attribute, see documentation[18]. All the tests use the initial configuration of the board; this means that the testing framework has to initialize the board in three different functions. We did not use parameterised testing because we have different board configurations for the different tests; therefore, we chose to initialize the board within every test.

Neighbour Finder

The test case "TestNeighbouringRectangularElectrodes" is testing the neighbour finder function for rectangular-shaped electrodes. The board is initialized from the JSON file "platform640v2.json", and the neighbours for the electrodes are found using the neighbour finding algorithm. Using the NUnit function "Assert.That()", we can test the neighbours of the electrode at index 0 of our container because we know that the neighbour array of this electrode only should contain an electrode with ID 2 and an electrode with ID 33. When running the test, we see that it does only contain the two specific electrodes, meaning that the test case passes.

8.2.1 Subscription Initialization

The test case "TestSubscriptionInitialization" is testing the subscription component of the initialize function. The board is initialized from the JSON file "platform640v1.json", and the neighbours for the electrodes are found using the neighbour finding algorithm. Using the NUnit function "Assert.That()", we test the subscriptions of the electrode at index 1 of our container. This particular electrode is tested because we know that after initialization, it should contain droplets with ID 0, 1 and 2 as subscribers, which it does, meaning that the test case passes.

8.2.2 Droplet Group Initialization

The test case "TestDropletGroupsInitialization" is testing the subscription component of the initialize function. The board is initialized from the JSON file "platform640v2.json", and the neighbours for the electrodes are found using the neighbour finding algorithm. Since the droplets with ID 0 and 7 are situated at the electrodes with ID 195 and 196, respectively, these droplets should have the same group number. Using the NUnit function "Assert.That()", we can test whether the particular droplets in the container in the droplet have the same droplet group, which they do, meaning that the test case passes. The tested droplets on the board can be seen in Figure 8.1.

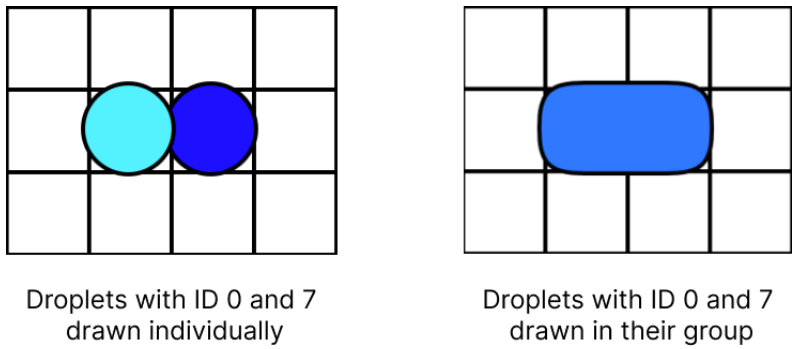

Figure 8.1: The droplets with ID 0 and 7 drawn on electrodes with ID 195 and 196 individually and as a group.

Droplet Movement

The test case "TestDropletMovement" is testing the movement models of the droplets. The simulator is initialized from the JSON file "platform640vCenterPath.json". The action queue of the simulator is initialized from the text file "center_path.txt". The test case checks whether the single droplet in this configuration has the correct initial position using the "Assert.That()" function. The simulator then runs 21 actions and checks whether the position of the droplet adheres to the expected position. The simulator then runs the remaining actions and checks whether the final position of the droplet adheres to the expected position. The positions match, meaning that the test case passes.

Droplet Split and Merge

The test case "TestSplitMerge" is testing the split and merge models of the droplets. The simulator is initialized from the JSON file "platform640vSplitMerge.json". The action queue of the simulator is initialized from the text file "split_merge_1.txt". The test case checks whether the container of the simulator contains two droplets after the initialization using the "Assert.That()" function. The simulator then runs 20 actions and checks whether the droplets have merged. It does this by asserting that the container only contains one droplet. The simulator then runs two actions and checks whether the droplets are in the midst of a split. It does this by asserting that the container contains five droplets. The simulator then runs 12 actions and checks whether the droplets have split into two droplets. It does this by asserting that the container only contains two droplets. The container does only contain two droplets; therefore, the test case passes.

8.3 Platform Testing

In order to test for the realistic behaviour of the simulation engine, we run several platform test examples along with real-life scenarios. This is done by reading a text file containing instructions which have been sent to the real platform and running these instructions on the simulation engine. With the real platform as a point of reference, we can compare the simulation of the platform with the real platform. The instructions executed on the platform can be seen online [19]. An example of a side by side comparison of a big droplet splitting across five electrodes in the simulation framework and the real DMF platform can be seen in Figure 8.2.

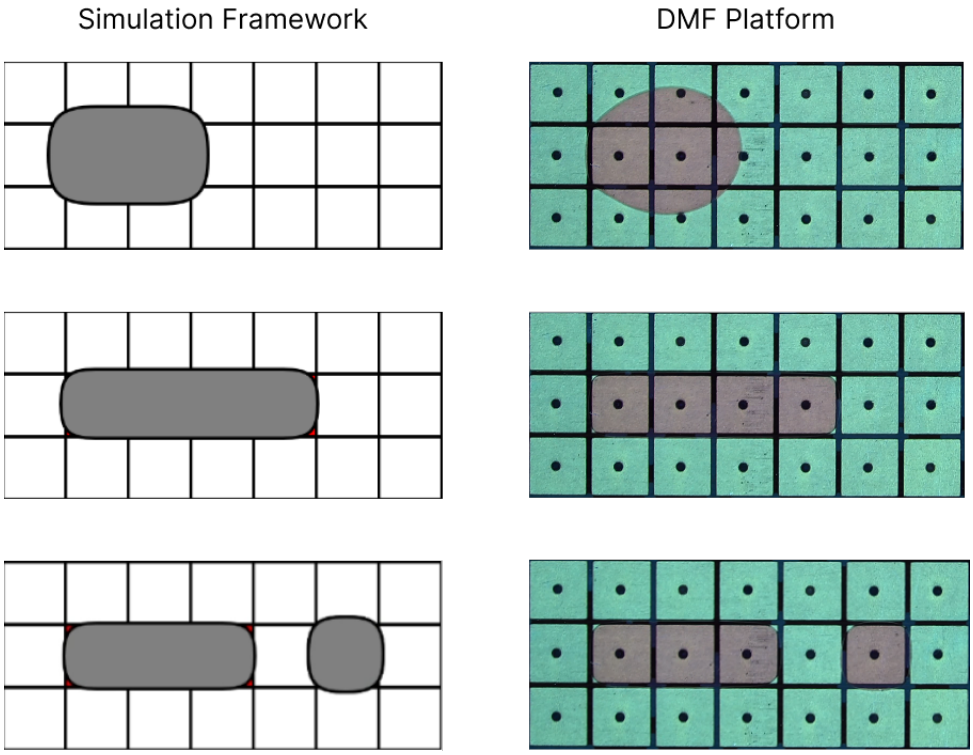

Figure 8.2: Droplet split in the Simulation (left) and the DMF Platform (right) respectively.

Figure 8.2 shows a static comparison, where snapshots from the simulation framework and physical DMF platform are compared. To showcase a comparison of the running simulation framework and physical DMF platform, we have constructed two dynamic comparisons in the form of videos. The first comparison is a simple movement sequence for a single droplet[1], although a simple sequence, it shows a clear correlation between the physical DMF platform and the simulation framework. A static capture of the comparison can be seen in Figure 8.3

[1]Movement sequence: `https://tinyurl.com/y2p7rk55`

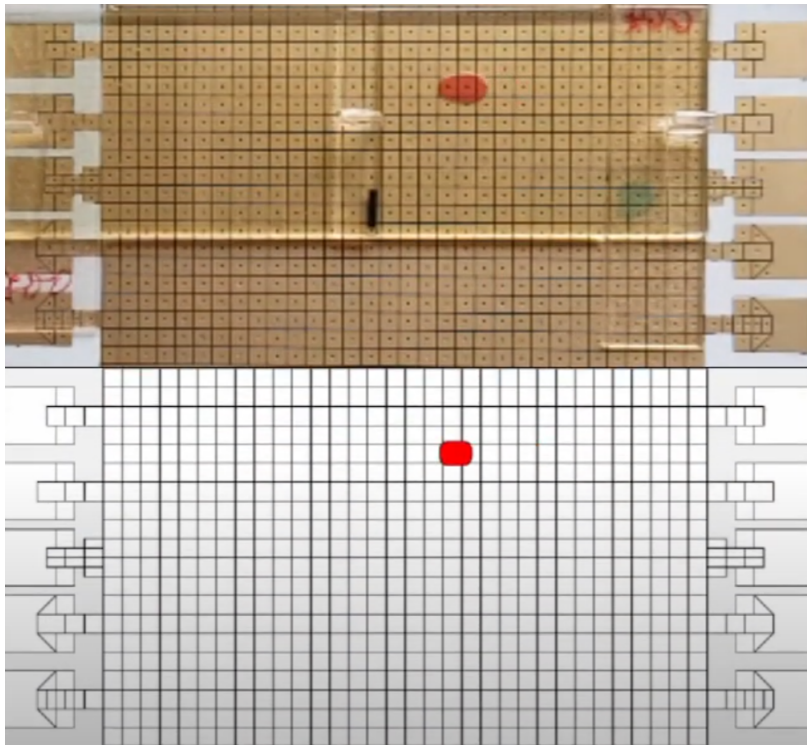

Figure 8.3: Image capture of the simple sequence comparison video.

The second comparison is created over a more advanced program that utilizes not only the simulation framework but also the dynamic communication between the simulation framework and the SimpleVM. The program dynamically spawns droplets, as if they were placed by a pipette, and moves them into a sensor; the SimpleVM reads the sensor data, and sorts the droplets based on the colour value read by the sensor[2]. The physical DMF platform is in an open system state in the video, and although the simulation simulates a closed system, we can see that the positioning is correct. For this comparison, we did not have access to the lab protocols run on the real platform, meaning that the testing configurations for the colour sorting are reconstructed from observations. As a result of this, the comparison has a slight error. However, it can still be used as a comparison metric, and we can see that the simulation and physical DMF platform are similar in execution. A static capture of the colour sorting video can be seen in Figure 8.4.

[2]Color sort: `https://tinyurl.com/3h8dxuyw`

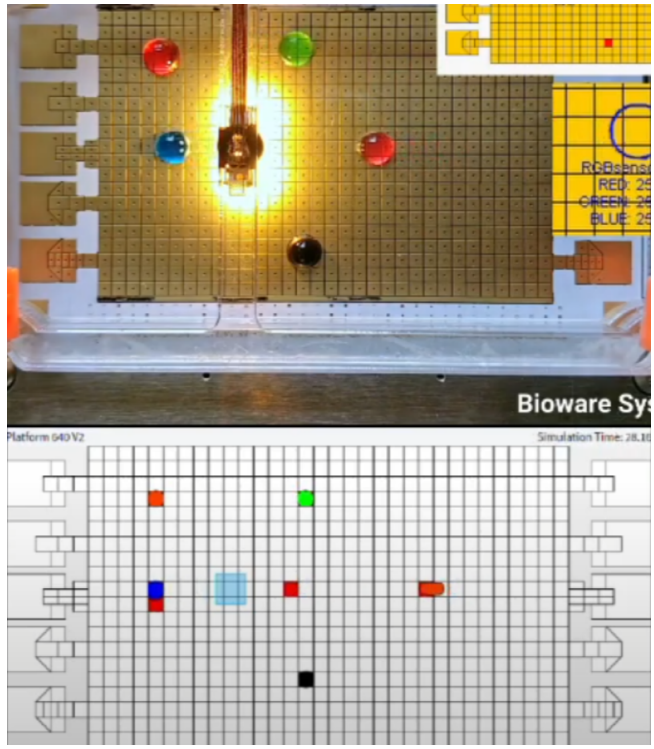

Figure 8.4: Image capture of the colour sorting comparison video.

We carried out additional platform tests, which are listed and shortly described.

- **Temperature**
  The purpose of this test was to check how the heater actuators affect the droplets. The expected behaviour was that after 90 seconds, the heater should go from $20°C$ to $90°C$, and if any droplets were positioned on the heater, they should get an increase in temperature, respective to the temperature of the heater. Conducting this test yielded the expected result.

- **Bubble spawn**
  The purpose of this test was to check if the droplets' model for spawning bubbles after a threshold temperature was implemented correctly. The expected behaviour was that a droplet above $90°C$ would spawn bubbles. If it decreased below $90°C$, the droplet would not spawn any bubbles. Conducting this test yielded the expected result.

- **Split and merge**
  The purpose of this test was to check the split and merge models of the droplets. The expected behaviour was that when two droplets moved into each other, they would merge into one droplet. If the electrodes on either side of the merged droplet were turned on, the droplet would split back into two separate droplets. Conducting this test yielded the expected result.

- **Droplet colour mix**
  The purpose of this test was to check how the droplets' colours change when they are merged. The expected behaviour was that when two droplets with distinct colours merged, their resulting colour should be a mix of the two colours. Conducting this test yielded the expected result.

- **Droplet movement limitations**
  The purpose of this test was to make sure a droplet would not move if it was either too small or too big. The expected behaviour was that if a droplet was smaller than the size of the electrode, the pull of a neighbouring electrode would not move the droplet. If the droplet was large enough, the pull from a neighbouring electrode would also not move the droplet but instead split out a smaller droplet from the big droplet. Conducting these tests yielded the expected results.

These platform tests were performed by running the cases in the simulation framework and closely inspecting the underlying data to observe whether the simulation framework's execution resulted in the expected behaviour.

A playlist with different configurations and programs running on the simulation framework can be found here: `https://tinyurl.com/yck8ayjb`.

8.4 Performance

Different optimizations were made to increase performance; one example is the way of traversing electrodes when finding neighbouring electrodes. This is elaborated upon in Section 5.4.2. In other cases, an optimization was needed to improve computation time, and therefore a new implementation was necessary. An example of this is the method used for initializing subscriptions. The optimization to the subscription initialization is explained in Section 5.3.1.

The most significant performance increase was related to data exchange between the simulation engine and GUI. When sending data from the simulation engine

to the GUI, the data needs to be serialized into a format that can be exchanged between the two, which can then be deserialized on the other end. The format used in this implementation is JSON. While sending the JSON data from the simulation engine to the GUI, we experienced quite significant time delays, and after conducting some time analysis on the data exchange, we found that the serialization of data was slow. Serializing the data in C# took around 200-400ms; this is quite a significant time span since the DMF platform executes in a matter of 50-100ms per execution. After researching this issue, we found that slow serialization is a known issue, and the only solution seems to be to swap the built-in serialization method to what is called a UTF8 serializer [20]. The UTF8 serializer provides a faster approach to serializing data and thereby offers the necessary performance boost in order to execute the actions without a noticeable time delay in the communication between the simulation engine and the GUI. By using the UTF8 serializer, the serialization time was decreased to around 30-45ms, a quite significant decrease.

Chapter 9

Discussion and Conclusion

This chapter presents the discussion, conclusion, and future works of this thesis. The discussion will tackle some of the limitations of the simulation framework, where the future work section will provide possible solutions to some of the issues that could be worked on if development were to continue. Finally, the conclusion will summarize and conclude the thesis, describing what was managed during the project's time span.

9.1 Discussion

Given the time frame and scope of this project, we had to choose between prioritizing our time on either creating test cases or further implementation of the simulation framework. We chose to prioritize further implementation since we found that for a bachelor thesis, this would be more valuable and result in a more well-rounded project. In traditional software projects, when utilizing unit testing, one would aim for test coverage of above 90%. With our unit tests, 90% code coverage is not reached; however, we created unit tests for core concepts of the simulation framework and carried out platform test observations and comparisons for many other aspects to ensure correct functionality.

Revisiting the specifications from Section 1.3, Figure 1.1, we can see that as

a result of our prioritization, we were able to accomplish all but two of our tasks from the MoSCoW analysis. We were unable to design and implement the chemical reactions, mainly because the topic's scope was greater than we first anticipated. That being said, with the modularity of the simulation framework, chemical reactions (in-droplet reaction kinetics simulations) could be added to the simulation framework at a later stage. We did not implement a dedicated translation unit, this was due to us not having access to the actual virtual machine, and the purpose of the translation unit was to convert virtual machine commands to commands in the simulation framework.

The MoSCoW analysis can be seen in Figure 9.1 where tasks highlighted with green were implemented, and tasks highlighted with red were not. Outside the scope were droplet pathfinding and board reservoirs; we did not implement these since the pathfinding is controlled within the virtual machine, generated from the protocol, and the reservoirs were not tested properly in the physical platform. The reservoirs are working in the simulation; however, we do not have a metric of comparison, and the behaviour could therefore be unexpected. This should be tested in the future and updated accordingly.

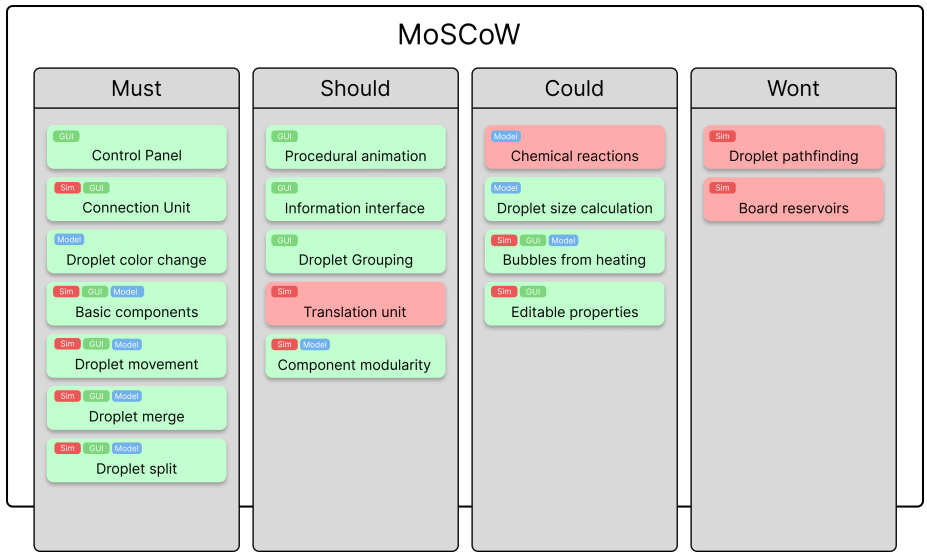

Figure 9.1: MoSCoW prioritization analysis, where green tasks are implemented and red are not implemented into the framework.

The simulation framework is built around the physical platform where an ITO glass is present on top of the board; this is a closed system, meaning that the droplets are squished between two layers, see Figure 2.2 for reference. This also

means that if the ITO glass is removed, creating an open system, the behaviour of the simulation varies. However, this does not mean that the simulation cannot be used to simulate the open system, the droplet positions should be equivalent, but droplet splits, and shape deformations are not possible in the open system, and the program should be built with this in mind.

When designing the simulation framework structure, we made a design choice to let every component of the structure communicate in real-time, providing the most realistic approach to how the physical platform behaves. In hindsight, this choice might have led to some bottlenecks, which resulted in a less realistic simulation, the main bottleneck being the time it takes for the simulation engine to compress data and send it to the GUI. Having the simulation engine and GUI only exchange a singular execution leads to a delay of approximately 45ms with every execution, this is not noticeable when running the simulation alone, but when conducting comparisons between the simulation and physical platform, the delay becomes noticeable after some time, where the simulation lacks behind. One could then rethink our initial approach of real-time communication between the components and realize that this real-time communication may not add anything of importance for the use of the simulation but solely to the thought of how it executes. With this in mind, a refactor of the communication could be made, allowing the simulation engine to run first and then use the GUI to replay the simulation in real-time. This topic will be touched upon again in Section 9.4.

After we finished the implementation part of this project, we found some possible alterations and additions that could be made to the framework; these are covered in Section 9.4

9.2 Conclusion

The scope of this project was to create a simulation framework that could simulate the behaviours observed on the physical DMF platform. The framework should support droplets, actuators, sensors, and should be modular, allowing for future components to be implemented into the framework. Furthermore, the framework should be integrated into an existing web-based framework, which only allows for client-side processing. During this thesis, we have presented our take on how to build a simulation framework that satisfies the specified scope. The simulation framework successfully simulates; the behaviour of droplets, actuators and sensors in a closed system; we have also implemented a simplified version of a virtual machine to showcase how such a virtual machine can interact with the components of the simulation. We have provided a graphical user

interface that not only allows for a visual representation of the simulation but also allows the user to control the flow, as well as properties, of the simulation dynamically. By creating a non-physical logical estimation of the physical behaviour, we have been able to create a simulation framework that allows for an execution time small enough to be run in real-time in the browser, solely on the client-side.

Through tests conducted by comparing similar programs running on the actual DMF platform and our simulation framework, we have been able to observe that the simulation framework does simulate the behaviours of the physical DMF platform. By comparison, we have been able to both compare simple programs, where only one droplet is moving[1], to more advanced programs where multiple droplets are moving in parallel, while the virtual machine directs them, by input from the sensors[2]. Through these tests, we have been able to conclude that the simulation framework is able to simulate the observed reality very closely. The simulation framework is, therefore, able to be used as a tool to test protocols before they are run on the physical DMF platform and to conduct experiments without any cost of failure.

9.3 Statement of Contributions

The three authors of this thesis, Alexander M. Collignon, Carl A. Jackson, and Joel A.V. Madsen, all contributed equally to the ideas and conceptualization of the simulation framework described in the thesis. The implementation of the simulation framework can roughly be divided into three parts, the graphical user interface, simulation engine, and models. The main ownership, meaning main responsibility of implementation and report writing, of the three parts can be split between the authors in the following way, Alexander Collignon had main ownership of the simulation engine, Carl Jackson had main ownership of the models, and Joel Madsen had main ownership of the graphical user interface. Main ownership is not equal to strict segregation, and in many events, we provided help to each other's domains. All other aspects of the thesis and implementation were created in collaboration.

[1] Movement sequence: `https://tinyurl.com/y2p7rk55`
[2] Color sort: `https://tinyurl.com/3h8dxuyw`

9.4 Future Work

This section presents some alterations that could be made to the project if we were to continue the development in the future.

In-droplet Reaction Kinetics

In-droplet reaction kinetics relates to the chemical reactions that can occur inside droplets when substrates are present. As an example, the reactions could be activated upon a merge of two droplets or when a droplet is heated to a specific temperature. If we were to develop the simulation framework further, the next big expansion would be in-droplet reaction kinetics since this would allow for experiments with substrates within the simulation. If implemented successfully, in-droplet reaction kinetics could allow biochemical programs, such as PCR (Polymerase Chain Reaction), to run in the simulator.

Surface Tension

The surface tension of the droplets is currently not taken into account when the droplets split. This means that the droplets will move or split immediately if an electrode is turned on. In reality, droplets could stretch across multiple electrodes even though these electrodes are turned off. An example of this can be seen in Figure 9.2,

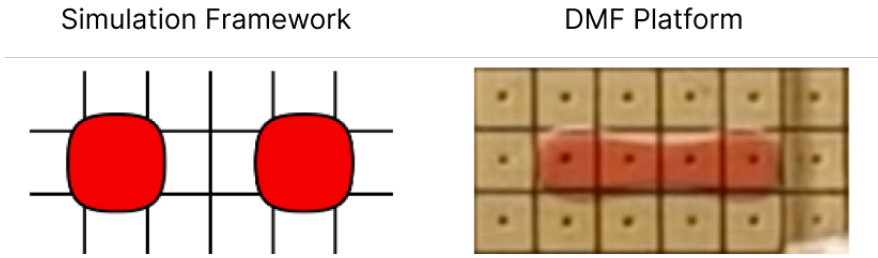

Figure 9.2: Surface tension of the real-life droplet causes the Simulation Framework (left) to behave differently than the DMF Platform (right).

The splitting in Figure 9.2 differentiates from the splitting in Figure 8.2. The

simulation is an abstraction of the real-life platform with high accuracy and deterministic behaviour. The simulation does not account for all the physical phenomenons that can occur during splitting, but future models could be developed that can handle such physical phenomenons. With the modular approach, this is fairly straightforward when the additional functionality has been identified and specified.

Split Volume

If a droplet is split across two electrodes, the droplet volume will eventually be distributed equally across the two electrodes. This is what the simulator simulates. When multiple electrodes are turned on, the volume is distributed equally across them. However, if an additional neighbouring electrode is turned on before the volume is distributed, the appearance will differentiate from the animation of the simulation. A possible addition would be to split a smaller droplet volume during a split and then distribute the volume of the droplet group across the electrodes over time.

Component Communication Overhaul

In our initial design, the components of the simulation framework, models, graphical user interface, simulation engine, and SimpleVM, were designed to communicate in real-time, meaning a single action is simulated and visualized before the next action can be simulated. However, this created an unforeseen bottleneck since the time it takes to send data from the simulation engine to the GUI takes approximately 45ms. This led to the simulation lacking behind over time when comparing it to the physical DMF platform. One way to solve this issue would be to remove the real-time communication, and execute every simulation step, then visually replaying the simulation afterwards. This would require the simulation to be rerun if properties were to change, but running 100 seconds of simulation takes approximately 4 seconds, so the rerun time would not be too big of an issue. Both approaches, real-time and replay, come with their own pros and cons, but it would be interesting to implement the replay approach to see if this method outweighs the bottleneck of the real-time method.

Bibliography

[1] Documentation Agile Business. https://www.agilebusiness.org/page/ProjectFramework_10_MoSCoWPrioritisation.

[2] Luca Pezzarossa. Bioassembly instruction set architecture. 2019.

[3] Viktor Anzhelev Tsanev. Front end for a digital microfluidic biochips simulator. 2021.

[4] Frederik Emil Schibelfeldt. A model-based simulation engine for digital microfluidic biochips. 2021.

[5] Paul Pop, Wajid Minhass, and Jan Madsen. *Microfluidic Very Large Scale Integration (VLSI)* . Springer, 2016.

[6] Paul Pop, Mirela Alistar, Elena Stuart, and Jan Madsen. *Fault-Tolerant Digital Microfluidic Biochips*. Springer, 2016.

[7] Documentation Stackoverflow. https://stackoverflow.com/questions/3722307/is-there-an-easy-way-to-blend-two-system-drawing-color-values.

[8] Documentation GeeksforGeeks. https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/#:~:text=BFS(Breadth%20First%20Search)%20uses,edges%20from%20a%20source%20vertex.

[9] GeeksforGeeks rrlinus. https://www.geeksforgeeks.org/minimum-distance-from-a-point-to-the-line-segment-using-vectors.

[10] MicrosoftDocs Microsoft. https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/classes.

[11] Greates Common Divisor Wikipedia. https://en.wikipedia.org/wiki/Greatest_common_divisor.

[12] Lauren Lee McCarthy. https://p5js.org.

[13] Gérald Barré. https://www.meziantou.net/optimizing-js-interop-in-a-blazor-webassembly-application.htm.

[14] JavaScript Window Object W3Schools. https://www.w3schools.com/js/js_window.asp.

[15] Gorilla Sun & Dave Pagurek. Webpage: https://gorillasun.de/blog/an-algorithm-for-polygons-with-rounded-corners.

[16] Documentation Radzen. https://blazor.radzen.com.

[17] Documentation Mozilla. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Manipulating_documents.

[18] Documentation NUnit. https://docs.nunit.org/.

[19] Carl Jackson Joel Madsen and Alexander Collignon. https://www.youtube.com/playlist?list=PLVOESxW3fkDQsGS8HsaVG53kN_xR_Biv9.

[20] Michael Shpilt. https://michaelscodingspot.com/the-battle-of-c-to-json-serializers-in-net-core-3/.

Appendix A

Appendix

A.1 GitHub

The code base of this project can be found on GitHub, using the following link: `https://tinyurl.com/424fs8z7`.

A.2 Definitions

This section will cover the definitions used throughout the thesis.

- **Pixel** - A pixel is the unit of metric used in this project. It is not similar to a pixel on a computer screen, but it is the standard metric for the components. An electrode is, for example, 20 x 20 pixels.
- **Neighbour** - A neighbour of an electrode is another electrode which shares any given side with the original electrode. If the two electrodes only share a corner, one pixel, see definition in SectionA.2, they are not defined as neighbours.

- **Shape** - A shape is referring to the shape of an electrode. An electrode can be shaped rectangular, or it can be shaped as an arbitrary polygon. The two shapes are represented in the JSON file as 0 and 1, respectively.
- **Deep copy** - A deep copy is a term used for a copy of an object. If a copy is "deep", it means that the elements in the copied object have no pointers to the elements in the original object. Therefore altering the values of the elements in the copied object does not impact the elements of the original object.
- **Container** - The container is a custom data type used to store all the components in the simulation engine.
- **Accumulating bubble escape volume** - The accumulating bubble escape volume is a field constructed to keep track of when a bubble needs to split from a droplet. This field is incremented if a droplet is above the threshold temperature but the time step is very low. This ensures that a bubble spawns when multiple of these short time steps have been executed.
- **Caller** - A caller is a term used for the object which is specific model or function is acting upon, an example is shown in Code Snippet A.1.

```
1    DropletModels.dropletSplit(container, caller);
```

Code Snippet A.1: Droplet split model called, here the "caller" is a droplet.

A.3 Datatypes

In this section we will be describing the data types used by the Simulation Framework, as well as the thesis.

The datatypes can be explained as follows.

- **Information** - The information datatype consists of a platform name, a platform type, a platform id and a size (x and y size).
- **GlobalVariables** - This datastructure holds global variables used in the simulation; it consists of a height (the space between the board and the glass on top), a maximum slider value (this is used to determine how far ahead in time you can skip), a room-temperature which is currently set to $20°C$ and a rectangular electrode size which holds the length of a side of the standard electrodes on the board.

- **Droplets** - A droplet consists of a name, a global id, a substance name, a position (x and y coordinates of the centre of the droplet), a size (x and y size), a string representing the colour, a float representing the temperature, a float representing the volume, an electrode id, an integer representing the group of the droplet, a double which is used for calculating the accumulating escape volume of the bubbles, a model order string array, a model order integer called nextModel and another model controlling integer called beginOfSensitiveModels.

- **GroupDroplets** - A group of droplets consists of a group id, a substance name, a string representing the colour, a float representing the temperature, a float representing the volume and a list of droplets' IDs.

- **Bubbles** - A bubble consists of a name, a global id, a position (x and y coordinates of the centre of the bubble), a size (x and y size), a boolean named "toRemove" (which is used when bubbles are removed from the simulation), an arraylist of subscriptions, a model order string array and a model order integer called named nextModel.

- **Electrode** - An electrode consists of a name, a global id, an electrode id, a driver id, a shape parameter, a position (x and y coordinates of the top left corner), a size (x and y size), a status (indicating if it is on or off), a list of corners (x and y coordinates relative to the position of the electrode), an arraylist of subscriptions and an arraylist of neighbouring electrodes.

- **ElectrodesWithNeighbours** - This datastructure is merely used for retrieving the neighbours of the electrodes if they are given in a JSON file. It consists of the same fields as the electrode datatype, but the arraylist of neighbouring electrodes is already populated.

- **Action** - An action consists of an action name, an action id and an action change. The action change is, for example, used as a parameter to set the electrode on/off status to the given number, which would be either 0 or 1.

- **ActionQueueItem** - An action queue item consists of an aforementioned action and a time. The action queue item is mostly used in a queue datastructure, where the time indicates when the action should be executed.

- **Actuators** - An actuator consists of a name, a global id, an actuator id, a type (e.g. "heater"), a position (x and y coordinates of the top left corner), a size (x and y size) and an arraylist of subscriptions.

- **Heater** - A heater extends the parent class Actuators; a heater consists of all the same fields as the actuator class with the addition of an actual temperature value, a desired temperature value and a power status value.

- **Sensors** - A sensor consists of a name, a global id, a sensor id, a string representing a type, a position (x and y coordinates of the top left corner), a size (x and y size) and an electrode id.
- **TemperatureSensor** - A temperature sensor extends the parent class Sensors; it consists of the same fields as the sensor class but with the addition of a float representing the temperature.
- **ColorSensor** - A colour sensor extends the parent class Sensors and consists of the same fields as the Sensors class but with the addition of three parameters, value red, value green and value blue, which constitutes the color that the sensor is reading.
- **Container** - The container is used to store all the forementioned data. It consists of an electrode array, a list of droplets, a list of bubbles an actuator array, a sensor array, an information field, a current time, a time step, an arraylist of subscribed droplets, an arraylist of subscribed actuators, an arraylist of subscribed bubbles, an object array, an outputs-array and an unclassified array.
- **Inputs** - This datatype is currently not in use; it consists of a name, a global id, an input id and a position (x and y coordinates).
- **Outputs** - This datatype is currently not in use; it consists of a name, a global id, an output id and a position (x and y coordinates).
- **Subscriptions** - The subscription datatype is currently not in use.

A.4 Class Diagram

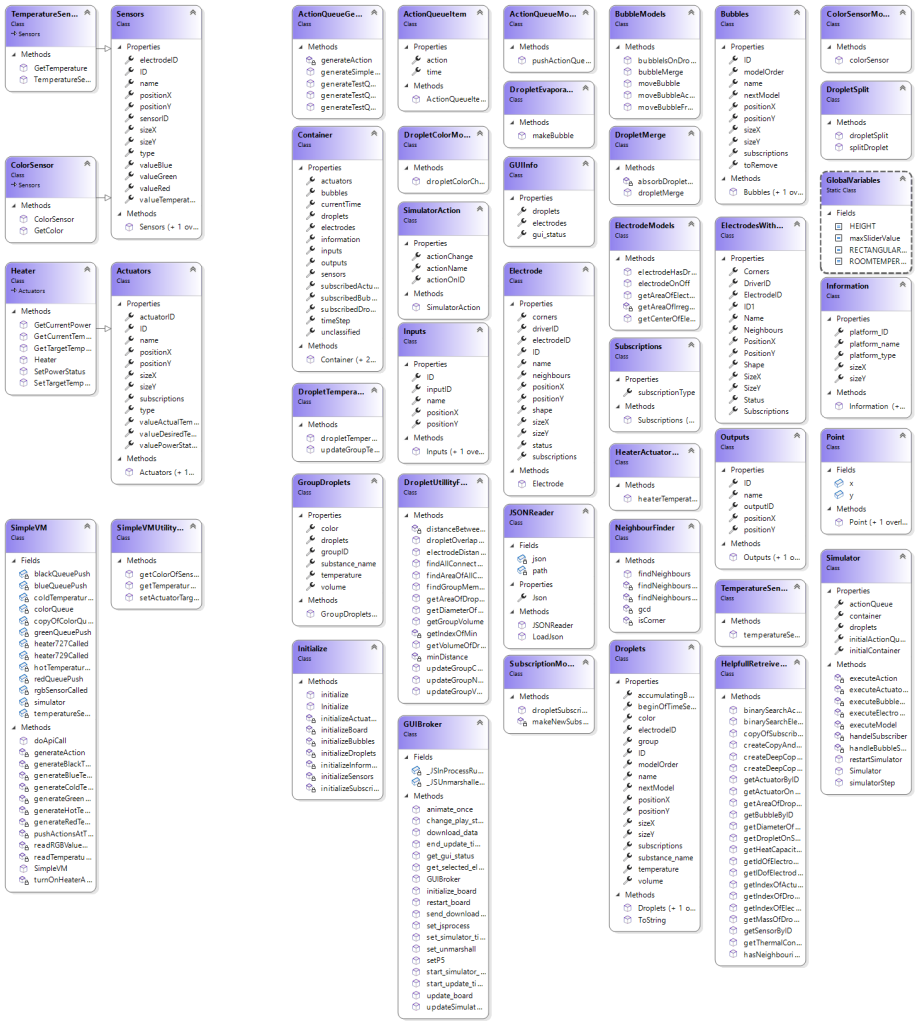

Figure A.1: Class diagram of the C# code for the simulation engine, models, and simpleVM.