Exploring Multiway Dataflow Constraint Systems for programming Robotic Autonomous Systems

Lars Oddne Ramstad Juvik

Master's thesis in Software Engineering at

Department of Computer science, Electrical engineering and Mathematical sciences,
Western Norway University of Applied Sciences

Department of Informatics,
University of Bergen

June 2023

Western Norway
University of
Applied Sciences

Abstract

Robotic Autonomous Systems (RAS) are robots able to perform tasks without human intervention. Today, we use them everywhere: from industrial usage as drones fertilizing crops or private usage as robot vacuums. Their capabilities are expanding rapidly, and humans trust them more and more. As they keep extending with features and new hardware, we must ensure that we use the best methods to program them. This is important to ensure the correctness of the software and to make developers able to reason about their code and program correct and safe programs. Safety is crucial in all systems, and Robotic Autonomous Systems is no exception, especially because humans are often close to robots, and care must be taken to avoid accidents.

A programming model that is not explored enough in such a context is Multiway Dataflow Constraint Systems (MDCS). In this programming model, you specify constraints between variables through methods. The constraint system will then satisfy the constraints by executing the methods in a particular order. Traditionally MDCS, whose semantics were specified in an article from 2021, has been used for programming Graphical User Interfaces (GUIs), where it can handle complex relationships between GUI elements[14].

In a drone, it is common to have many inputs and outputs: GPS, gyroscope, distance lasers, propeller motor, and actuator, to name a few. As the different combinations of inputs and outputs grow, along with new features, we can expect complex relations between inputs and outputs. Following this, we can draw similarities to the complexity of relations between GUI-elements. Knowing there are similarities to be found, exploring programming Robotic Autonomous Systems using MDCS and researching the applicability could be worthwhile.

This thesis will explore how MDCS can be applied to physical systems by creating a virtual simulator and adding a drone with a custom autopilot created with MDCS. Using an implementation of MDCS called *HotDrink*, the system created in this thesis research how MDCS can be applied to a robotic system. The simulator will utilize a physics- and rendering engine so that we can observe how the MDCS-powered system handles different environments. Adding more features to a robot expands its actions and gives us new edge cases and potential problems. As the relationship between these different parts can resemble relationships between elements in a graphical user interface, it is worth exploring if using MDCS for such an autopilot can be beneficial. This thesis will focus on the developer side of the process of creating such a system, and the research questions focus on the developer's advantages and disadvantages of using such an approach.

Acknowledgments

First and foremost, I would like to thank Magne Haveraaen for being an excellent supervisor, teaching me about Multiway Dataflow Constraint Systems, and enthusiastically answering all of my questions.

I also want to thank Mikhail Barash for his explanations of the workings of MDCS and Knut Anders Stokke for giving me tips and tricks on how to program in HotDrink MDCS.

Thanks to Jaakko Järvi and the *HotDrink*-team for their contributions to the *HotDrink*-library[15]. The thesis would not exist without it.

I want to thank both Magne Haveraaen and Jaakko Järvi again for their contributions to Multiway Dataflow Constraint Systems, the foundation of this thesis.

This thesis is built upon a pre-made template called "HVL Master Thesis Software Engineering Template" [17].

Contents

List of Figures

List of Tables

Glossary

autopilot A system for controlling a vehicle without the need of an operator. An example of this would be a drone's ability to hover without input from a humar.

companion computer In the context of drones, a computer that sits on the same drone as the autopilot hardware. The companion computer can aid with more complex and time-consuming calculations and then sends it to the autopilot so that the autopilot can focus on more time-critical computations.

digital twin "A virtual model designed to accurately reflect a physical object"[31].

domain-specific language "DSLs are small languages, focused on a particular aspect of a software system"[11].

HotDrink A JavaScript library that implements MDCS, such that we can use MDCS concepts in JavaScript-enabled environments..

LiDAR A laser that can measure distances.

mode For this context, it refers to an operation setting the drone can do, such as "hovering" or "object avoidance". Each mode implemented is using MDCS.

rigid body In the context of game development, a common term for an object that can be affected by physics.

separation of concerns "A principle used in programming to separate an application into units, with minimal overlapping between the functions of the individual units" [28].

Acronyms

CAGR Compound Annual Growth Rate.

GUI Graphical User Interface.

MDCS Multiway Dataflow Constraint Systems.

RAS Robotic Autonomous Systems.

Chapter 1

Introduction

Robotic Autonomous Systems (RAS) are becoming increasingly popular, and their use cases are expanding. They can react to the environment on their own, and they can be handed tasks that can be done without human intervention. They inhibit many fields of operation: agriculture, medicine, and military, to name a few. They have found their way into ordinary households, and owning a quadcopter that can fly from A to B by itself or a robot vacuum cleaner is no longer uncommon. Physical systems such as these are becoming increasingly common, and we increasingly trust them. Together with other technologies, they can solve many complex problems in different areas. With this surge in popularity, we need efficient ways to program them and make them adhere to their goals safely and predictably. It is essential for autonomous drones to have safe systems and some degree of predictability as they react to the environment independently. We don't want any erratic or unexpected movements. An example of this expansion of RAS is in a subset of these robots; the drone. The market for consumer drones is vast and expanding. In 2022, the global consumer market for drones was valued at 4 120,8 million USD and was believed to have a growth of 13,3% Compound Annual Growth Rate in the period from 2023 to 2030 [9]. This increase is significant, and if the prediction is correct, it will ensure drones stay under heavy development for many years. Continued growth also suggests that humans will continue to develop new features and systems and perhaps rely on them more to do increasingly essential jobs.

An important focus must be made on designing robust and safe systems for RAS, primarily since they can operate close to humans, animals, and structures and can cause severe harm. There have been accidents revolving around drones, and they can cause severe damage. One example is from 2021 when a drone collided with a plane - luckily, no one was hurt, but the aircraft suffered "significant" damage [20]. This is scary, as losing control of a drone is a bad situation that can be hard to stop or control. Another example is from 2017 when a drone collided with an army helicopter, and a bit of the drone was later found "right at the bottom of the main rotor system" - none of the paratroopers was hurt [12]. Scary situations like these remind us to be aware of the potential damage drones can cause. Drones can cause scary situations and cause extensive damage. This can happen because of operator error, or they can suffer technical failures and

Figure 1.1: A flying drone. [3]

drop down from the sky. Designing safe systems in the context of RAS means designing them to operate safely on their own and assist the human if they were to take control of the robot (e.g., active object avoidance while an operator is controlling the drone).

A clear understanding of the code behind the systems is essential to design safe and robust logic and ensuring the system's correctness. The code must be correct, reasonable, and make logical sense. The systems should not be full of "spaghetti code" - code jumbled together, with no logical separation of concerns or structure. For the development process, developers must ensure their code is correct and efficient enough to operate the system fast enough for its intended tasks.

Every type of RAS can potentially cause harm and do actions it was not supposed to do: a drone can fall from the sky, a robot lawnmower can go out of bounds, and a robot vacuum can tilt over a vase. Accidents happen for different reasons, and the system, user, or both are at fault. It is hard to catch every edge case when designing these systems, as extremely many unintended situations can occur. A drone can crash into a structure for many reasons: poor piloting, drained battery, hardware fault, software fault, pilot unsure of the rules, etc. Whatever the reason, the programmer should always ensure the system is as fail-proof as possible.

The project should be kept structured and understandable, especially when the project grows to fit many different features. For instance, a modern robot vacuum cleaner claims the following features, plus more [7]:

- High Precision Map

- Map Saving
- Zone Cleanup
- Real-Time Robot Location
- Voice Control Adaptive Route Algorithm
- LDS Laser Navigation Auto Mop Washing
- Auto Mop Drying
- Auto Tank Refilling
- Auto Dust Emptying
- High-speed sonic mopping
- High Precision Map

A system containing this many features is set to be of significant size. These features can also contain complex logic, and it is important to keep researching if more suitable programming models exist to design these systems, such as MDCS.

Figure 1.2: A robot vacuum cleaner.[4]

When systems get big, they can quickly get overwhelming unless it is structured in a certain manner. Take drone autopilots, for instance: the repository where the code is stored can have many lines of code. The PX4 repository, which is an open-source autopilot for drones, has around 970k lines per 11. may 2023 [1]. This speaks of the volume of code that is required to operate such an autopilot.

This thesis focuses on programming an autopilot for a specific vehicle under the Robotic Autonomous Systems-umbrella; the quadcopter. It may be the most popular RAS, and its use cases span many sectors. In this thesis, I notion the quadcopter as a **drone** for simplicity. A drone is a more common term, and we often talk about quadcopters when speaking of drones.

The idea is to program an autopilot for the drone in a virtual setting. For this, we will need four main parts; a web application, a physics engine, a rendering engine, and an implementation of MDCS.

1.1 Web Application

The web application is written in React, which is a popular library for composing user interfaces [25]. The choice landed on React, as it is prevalent, and this typically means it has a lot of community support and libraries made for it. This makes it easier to seek help, as we can be confident it has a wider audience, and many common questions have been asked and resolved. It also fits well with the rendering and physics engine, as presented below.

1.2 Physics Engine

For the physics engine, we need a library that can simulate a physics world and be able to act forces on physical objects within this world. It should, as closely as possible, simulate the real world. We need to create a world with ground (so the drone can crash down on the ground instead of falling forever) and a drone that is affected by forces like gravity. The library used in this project is *react-three-rapier*, which acts as a wrapper around the *rapier* library so that it can easily be used with React. The library will be presented more in later chapters [27] [24]. We can simulate a world with the physics engine sorted, although we can't see it yet. We need a rendering engine and connect it to the physics engine to observe the world visually.

1.3 Rendering Engine

A rendering engine is responsible for displaying objects to the user. When creating a simulator like this, it is essential. For this project, the rendering engine is *react-three-fiber*, which acts as a wrapper around *three.js*[23] [30]. As with *react-three-rapier*, this library also acts as a wrapper, so it's easily integrated with the web application written in React.

1.4 MDCS

Multiway Dataflow Constraint Systems is "a programming model where statements are not executed in a predetermined order"[14]. Multiway Dataflow Constraint Systems automatically oversees the constraints and values specified and calls certain functions when a value is changed to uphold the constraint. For instance, if MDCS were used in a drone, the constraint system could specify a constraint between its height and its upward thrust to keep its height stable ("hovering").

1.4.1 Implementation

This programming model is implemented in a library called *HotDrink*, which I will use for this exploration [15]. It is a JavaScript library that also comes with

TypeScript types, which means we can use the library in a typed manner. The library also provides a domain-specific language that can be used. In this thesis, it is not used, as the focus was on presenting the system in a typed manner, where it is clear which objects are created and how we can create them.

1.5 Context and Approach

1.5.1 Context

In software engineering, it is important to keep widening our vision of available technologies and go deep into them to gain a new understanding of our technological climate. Exploring new programming models fit into this mission, and hopefully, the exploration presented can expand our knowledge of MDCS. The focus of this thesis is not on the raw efficiency of the system but more on exploring how this system would interact in the real world with real physics constraints and what the system's architecture looks like. A comparison between this code (using MDCS) and code not using MDCS will be presented in chapter 5. This project is worthwhile to explore because it uses MDCS in a physics context, but it has typically been used in a graphical user interface context[14].

The thesis was raised from a problem of a drone navigating around an active wind turbine, which requires particular safety to prevent damage to the wind turbine. To do this as safely as possible, there are many observations and decisions to handle. As there are many complex relationships and actions to keep track of, a system with MDCS could be made, as it can already be used to handle complex relationships in GUI programming. As modern autopilots are advanced and can perform many different tasks, the focus shifted to exploring whether another programming model could suit robotic systems. This original problem is further presented in chapter 2.1.

1.5.2 Approach

Multiple approaches were considered to implement the idea of getting a working (simplified) autopilot on a drone. The most important part was to get an implementation of MDCS to work with. This has already been worked on in a JavaScript library called HotDrink, which implements MDCS[15]. The library has previously been tested with Graphical User Interfaces. Where you in GUI can just "set" fields to satisfy a constraint, in the physics world, you need to "push" objects - you can't just set a new position - this would be teleportation. Therefore we can define constraints between desired properties of the drone (e.g., the altitude to hover at and the force output of the drone) and let the constraint system handle the calling of different methods.

The approach started with reflecting on using an actual drone, using MDCS, but this raised several issues. First of all, testing the software on a physical drone is more complicated than programming software that runs on your computer, and any bugs can prove costly as they can cause the drone to crash. Second, since we are using *HotDrink*, we need a way to run JavaScript on the drone. This could be done with a companion computer, perhaps running Node, which allows for running JavaScript outside a browser. But again, this adds to the

complexity of testing the approach using MDCS - we are really only interested in how the architecture and implementation look on such a system, and it is therefore not crucial to have a physical drone. And third, a physical environment is practically impossible to recreate. This means if we get the drone to run in some conditions, there is no guarantee that it will work again under different conditions. It makes it hard to re-create scenarios, especially cases that only occurred during a particular environment. An example of this would be if a physical drone suddenly dropped to the ground when a gust of wind came at it. After testing this again and receiving a similar gust of wind, the drone might not be tipped over again. This lack of replicability makes it hard to pinpoint what went wrong when it tipped over the first time. This issue is solved with a deterministic physics engine, where you can get the exact same situations, given the same initial state and steps.

Soon, the idea of using a digital twin arose and seemed more and more like the better idea. If a simulator could be made to put a drone in, we could control the drone and the environment. As a bonus, we could use the HotDrink library directly if that simulator supported JavaScript.

At first, when the digital twin idea came to life, testing on Unity was done, which is a popular game engine. It works great as a physics and rendering engine and provides excellent tools to quickly build scenes with physical objects that can react to the physical world (rigid body). Although this seemed lovely, the big problem was the interaction with the HotDrink library. It is possible, but it is more cumbersome than just using the library in a project that supports JavaScript directly by default. To use JavaScript in Unity, one must set it up to make it callable from C#, which is the language Unity uses. It is doable, but compared to the ease of developing an application in the browser, it was enough to favor developing a new simulator using pre-made libraries for graphics and physics. The approach, therefore, landed on creating a digital simulator in the browser, where the end-user could see a drone using *HotDrink* control itself, made possible by already-made libraries for physics and rendering.

1.6 Problem Description

Robotic Autonomous Systems are getting increasingly complex: modern systems can have many sensors, actuators, and motors. Depending on these sensors' values, the system probably needs to adjust its behavior often through its actuators and motors. While these more complex drones benefit the users, what about the developers? Can a new methodology for programming result in a less complex codebase, with code that is easy to reason about, more maintainable, and helps with separation of concerns?

With all of this presented, here are the research questions:

- Does adding MDCS to an autopilot system yield benefits in understanding and getting an overview of the code?
- Are there clear advantages or disadvantages to using MDCS when programming RAS?
- How does the code compare to more "traditional" autopilot code?

1.7 Methodology

This thesis aims at testing the development of MDCS-systems on Robotic Autonomous Systems. The methodology's core is to incrementally create more complex demos so that, in the end, a reasonable enough judgment can be made if MDCS is advantageous in a robotic vehicle from the developer's point of view. The evaluation of the advantages, where the comparison is against how a traditional implementation without MDCS would look like, will be on general readability if it improves understanding and correctness of the code and if the abstractions MDCS provides are helpful in this project. There will also be a minor weight on lines of code. More lines of code are not inherently bad, but it still matters if lines of code greatly outweigh the traditional approach and no extra benefits are given.

To have something to evaluate, a simulator will be constructed, where we can add drones with an autopilot implementation using MDCS. This simulator fits inside a web application created with React. In the web application, the site will be laid out with an intuitive demo-chooser so that the user can easily switch demos and different pages for other information. On the main window on the main page, a rendering engine will display the selected physics world. In this window, the user can also choose different values to provide to the autopilot to observe the effects this has. When the overall system is complete, and we can see the behavior of the MDCS-autopilot, we can test different scenarios, control systems, and forces to see that the autopilot works as intended. Inside this simulator, we can test multiple setups and evaluate how MDCS works for Robotic Autonomous Systems.

The thesis will present simple demos of MDCS-controlled drones and the code that powers them. The demos start simple before building up to more complex systems by combining modes for the drone, each mode made with MDCS. This way, a good understanding of how the code changes with increasing complexity can be observed. Figure 1.3 shows how the simulator looks.

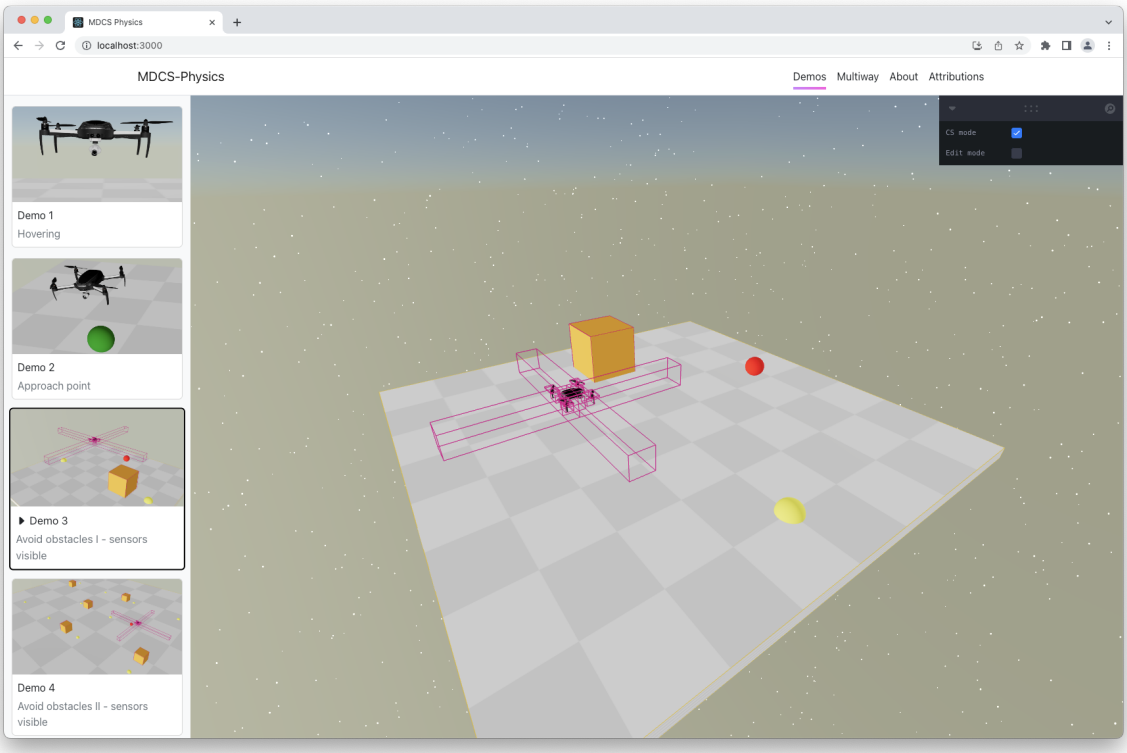

Figure 1.3: A drone using its MDCS-system to navigate.

1.8 Outline

Chapter 2: This chapter presents some background to prepare the reader to examine autopilot systems. The original problem of this thesis is also explained and put into the context of this thesis.

Chapter 3: In this chapter, the application is presented, along with relevant code to display their use.

Chapter 4: The use cases of MDCS in robotics are laid out along with potential use cases that could be developed.

Chapter 5: In 5, the project's findings and how it could fit in the context of software engineering are discussed. I will also discuss how some of the autopilots could be developed in pure React.

Chapter 6: Here, the paper will present related work and why it is of interest to this thesis.

Chapter 7: In the concluding chapter, the discussion chapter will be distilled to present the ultimate findings of this thesis.

Chapter 8: There is more work to be done in the context of the thesis, and this chapter lays this out to the reader.

Chapter 2

Background

This chapter presents critical topics that form this thesis and how they all tie together. Core technologies information that is used to create this project, such as JSX, is also presented so that the reader is acquainted with the technologies used.

2.1 Original problem

The thesis was raised from a problem regarding how a drone can navigate safely around an active wind turbine to take measurements of turbulence. This requires particular accuracy and safety measures to prevent damage to the wind turbine. Many events might occur, and there are a lot of factors to consider when designing drone systems. To name a few, the system needs to check:

- What is the wind speed where the drone is?
- How close is the drone to the wind turbine according to GPS?
- How close is the drone to the wind turbine according to distance sensors? Can the sensors "see" the wind turbine?
- What should happen if the GPS tells the drone it is 100 meters away from the wind turbine, but the distance sensors read 20 meters away from the wind turbine?
- How is the wind affecting position/orientation/speed?
- What should happen if the drone gets too close to the wind turbine?

There are many considerations to be made when designing such systems. The autopilot should also consider what actions to take if something goes wrong. When something goes very wrong on a mission, the autopilot should trigger a "fail-safe" mechanism. This mechanism is responsible for aborting the task and trying to move the robot to a safe location for later retrieval. In severe cases, the fail-safe mechanism could also be to cut the power and stop the vehicle immediately in order to prevent potential damage. These fail-safes can also be complex. If the drone is flying too close to a wind turbine in the ocean,

for instance, and the fail-safe mechanism is triggered, what should the fail-safe mechanism be? Usually, that could be to return to where the drone took off from. But what if the wind turbine is out in the ocean, and the drone takes off from a drifting boat? Then the drone would need to be updated with the boat's location or find another course of action.

All in all, there are many different actions the drone could take based on the observations it makes. This leads to a complex relationship between input and output. Other missions might have completely different outputs from the same inputs; it is all dependent on the current mode. We can draw a parallel from these missions to complex GUI forms. GUIs can have many different types of inputs, e.g., number fields or checkboxes. Updating one of the inputs might lead to an update in another element. For instance, if the value of a number input is over 1000, a checkbox somewhere else on the page should be checked. This is not too difficult, but when the logic gets more complex, and a relation between two elements is dependent on other elements, the relationship might be hard to model clearly. With this in mind, we can draw a parallel to MDCS, which can be used to develop GUIs with complex relations over the elements. This is why MDCS could be a valid option for designing autopilots to handle these complex relations between input and output.

2.2 Robotic Autonomous Systems

Robotic Autonomous Systems comes in many forms: robot vacuums, delivery robots, and quadcopters, to name a few. There are many different kinds of robots, and they span many markets and sectors. They are becoming increasingly common, and many private and business consumers use them daily. They can use them to vacuum the floors with a robot vacuum or shoot footage for a social media page with a drone. Today they are considered useful tools, and many businesses depend on them daily.

The market has been forecasted (in the period from 2023 to 2030) to have a growth of 13,3% Compound Annual Growth Rate[9]. This speaks of the interest in drones, and with the increase in interest and popularity, we must continue refining the products and services to ensure quality systems are being created. Refining is essential when the market reaches broader usage and our society increasingly adopts the product.

For large and complex Robotic Autonomous Systems, adding new features increases the risk of something going wrong. With Robotic Autonomous Systems, there is always the risk of errors, which can cause damage. Building safe and robust systems to prevent this is essential, as safety is the top priority. Building safe and predictable systems is becoming increasingly more complex when presented with many different inputs and means of output.

Writing code to handle and compute the combinations of these inputs, outputs, and edge cases can easily lead to code that is hard to read and tangled up; "spaghetti code." There are also many edge cases and different actions that need to be executed under specific circumstances, which again can contribute to this "spaghetti code" if not implemented carefully. When programming these systems, separation of concerns is an important concept, and it can help reduce

the disorder in code and make it more readable and understandable, which is very important. Maybe MDCS could contribute with separation of concerns and make the code more clean and understandable. One of the perks of using MDCS is that it can be adopted incrementally, connect to parts of the drone where desired, and automatically handle the behavior. However, it might be more beneficial and understandable to use if architecture choices have been made that favor the usage of MDCS.

Systems like MDCS can be connected to RAS. It could be connected to a low-level hardware API on the RAS, like on a companion computer on a drone that communicates with the firmware. Then, the constraint system could automatically control some part of the drone to uphold the constraints, e.g., stopping the drone if the drone is heading to an area specified as forbidden by the user.

2.3 Multiway Dataflow Constraint Systems

The heart of this drone system will be in the Multiway Dataflow Constraint Systems-system, which is specified in a paper from 2021 by Magne Haveraaen and Jaakko Järvi[14]. It is "a programming model where statements are not executed in a predetermined order. Rather, individual methods are selected from specific method sets and then executed to achieve a desired global state." [14].

The system needs to be given three main components to work: variables, constraints, and methods. In HotDrink, this can be put in a component: a container for clustering together related elements so that they can be treated as a single unit and added or removed together [18]. The constraint system works by having multiple constraints and ensures to satisfy these constraints using developer-defined methods. A set of methods define each constraint, and the methods should satisfy the constraint they are a part of. These constraints are applied between variables. If you have three variables, a, b, and c, you can define a constraint between them by providing a set of methods. If either of those variables changes by the user or the system, the system will automatically fire the methods belonging to the constraint to satisfy it. The planner selects the methods, one from each constraint's methods set, and can use the history of the global state's updates to determine the appropriate method to execute[14]. The methods are then executed to reach a global state that is desired[14].

Suppose you have variables, constraints over the variables, and implemented methods to solve the constraint between the variables. In that case, you have a valid component consisting of constraint(s) that can be used in a constraint system. You can then add the component to the constraint system, and when a variable is updated, the system will fire the different methods to satisfy the constraints. **The programmer is responsible for programming methods that will solve the constraint between the variables**. The constraint system's job is to fire the correct methods in a specified order.

We can build complex systems and behaviors from MDCS, for instance, graphical user interfaces. According to the paper by Haveraaen and Järvi, Multiway Dataflow Constraint Systems is deemed better to program graphical user interfaces than the more traditional event handling programming[14]. Graphical

user interfaces can have complex relations, and updating one input field can lead to multiple other fields being updated as well. Doing this all manually can be complex, and there is a danger that it could lead to "spaghetti code." We can find similar complexity with event handling in a physics context. Consider a scenario where a drone controller by a human operator suddenly lost contact with the radio controller. This could fire an event, which could lead to multiple other actions happening: compute a point to return to, decide the speed to use, compute if the battery is full enough to return, etc. One event can propagate to multiple other events happening, and complex relations between elements are formed, e.g., the rotation speed of the propellers, and the vector to a computed destination. Some relations are easier and are formed more naturally. A more natural and simple constraint would be that the speed should follow a log_{10}-function, where the input is the distance from the destination point. The ability to model real-life situations into constraints makes it worthwhile to explore using Multiway Dataflow Constraint Systems to solve the problem of executing the responses of all of these events manually. Instead, the system can choose which methods to execute.

MDCS is an exciting candidate to program such a system, significantly since it differs from other papers' domain focus, which is user interfaces [14][29]. In a user interface setting, changes can be reflected instantly. One could instantly set a text field to contain the text "Hello" if a user clicks a particular button. This does not apply directly to a physics context. For example, in a physics context, one cannot instantly set a drone's position - this would be teleportation, which is impossible as of now. In the real world, to get from point A to point B, you must apply forces to push an object to point B. This force and objects can be all sorts of things: a hand delivering an apple to another hand, a jogger running from a house to a park, or a drone using its propellers to propel forwards towards a point. This needs to be reflected in the constraint system. Even though I can technically set the position of a drone instantly using the physics engine, this will not be done, as it would go away from the real-world similarity we are trying to approach.

2.3.1 Examples

If you define multiple methods in a constraint, it means that you can use those variables to compute each other. An excellent example of this is the constraint $A = B - 1$. If you know A, you can compute B; if you know B, you can compute A. It goes both ways and can therefore be used in a *multiway* manner.

Consider this code of a constraint system, written in *HotDrink*'s domain-specific language:

```
/* Builds a system to satisfy the relation a = b - 1 */
var a = 1, b = 1;

constraint C {
  component {
    method1(a -> b) => a + 1;
    method2(b -> a) => b - 1;
  }
```

```
}
```

In this example, the dataflow goes both ways: the system can compute b from a, and the system can compute a from b. This code makes it clear and concise how the constraint should be structured, and the developer provides methods to solve the constraint between a and b. However, consider another example in a physics context: we have a drone with a GPS-variable `GPS`, a desired altitude variable (`desiredY`), and an output force variable that represents force from the drone perpendicular to the ground (`outputY`). The goal of the autopilot is to fly above a minimum height (accelerate upwards) and descend if it is ten units over the min-height (apply less force). Consider the following code written in *HotDrink*'s domain-specific language to achieve such hover-functionality. Be mindful that the code is simplified and would not be a good implementation in a real-life example. But for this example, it is sufficient.

```
var GPS = {x: 1, y: 1, z: 1};
var outputY = 0;
var desiredY = 3;

constraint C {
  component {
    method1(GPS, desiredY -> outputY) => GPS.y < desiredY + 10 ? 10 : 1;
  }
}
```

In this example, only one method is added to the method set of constraint `C`, but there are three variables. The last example had an equal amount of variables and methods. Why is this different now? To understand this, we should look at each input and output combination for each constraint and understand how they are computed. There are three combinations we could have:

Case 1: We are given `GPS` and `desiredY`, and should return `outputY`. This is what the example did; it works since we can compute the required forces from the input. We know that if the GPS y-coordinate (`GPS.y`) is less than the `desiredY + 10`, we should return more force and less force if that is not the case.

Case 2: We are given `outputY` and `desiredY`, and should return `GPS`. This does not work, as the GPS cannot be computed from these two variables. We can only figure out other information, such as which value is larger. This does not help find the GPS position; we can only tell if `GPS.y` is under or above the threshold.

Case 3: We are given `outputY` and `GPS` and should return `desiredY`. In this example, it does not make sense to compute the desired altitude, as the user should only set it. Where the drone is, or what its current force output is, should not change that. This is a read-only variable from the constraint systems point of view.

Since it does not always make logical sense to implement methods for each input/output combination in a physics context, it makes it harder and more counter-intuitive to utilize the multiway pattern of MDCS. However, we can still design the systems using MDCS, and some other scenarios could be easier

to implement multiple methods in, or do so by re-structuring the example code to include and compute different variables.

2.3.2 Key Terms

Here is an overview of the key terms regarding MDCS I will use in this thesis.

- Constraints - a condition you can define between multiple variables that should be satisfied, e.g., `a + b = 1`. Constraints contain a set of *Constraint Satisfaction Methods* (called **methods** from here on) defined by the programmer. When the variable updates, a method made by the programmer is called to ensure that the constraint is still valid. "constraints between variables are expressed as sets of functions". [19]
- Constraint System - the system that holds the variables and enforces methods to try to satisfy the constraints.
- Component - "HotDrink's abstraction for grouping variables and constraints, so that they can be added and removed from a constraint system as one unit" [18]
- Variable - In *HotDrink*, we can subscribe to variables and run custom functions when the value is updated. We also get some details about the event; see table 2.1. The programmer can also set values through a reference to the variable or through the constraint system. "Changing a variable's value invalidates all the constraints that the variable belongs to."[18] This leads to a lot of in-validations as I will need to update the position of the drone as often as possible in the simulation to get the best/most recent calculations from the MDCS-system.

2.4 HotDrink

HotDrink is a JavaScript implementation of MDCS and allows us to use constraint systems in JavaScript-capable environments [15]. *HotDrink* allows us to use a constraint system in our applications in an easy-to-use manner and has been used to create Graphical User Interfaces. It provides types in case you want to use TypeScript, which this project used, and supports templates if you want to use HotDrink in a more pure domain-specific language-form. There is also *hotdrink-rs*, which is an implementation of MDCS in the Rust programming language made by Svartveit [29] [16].

The library *HotDrink* is made for implementing Graphical User Interface (GUI) declaratively, but in this thesis, it will be used in the context of physics[18]. Using TypeScript with it, we gain access to multiple types that will be helpful: `ConstraintSystem`, `Component`, and `ConstraintSpec`, to name a few. We will also use the method for adding variables to a component through `component.emplaceVariable(...)` and for adding constraints to a component through `component.emplaceConstraint(...)`.

HotDrink supports subscribing to a variable value, so you can define certain functionality to be run when this happens. This is more efficient than fetching the newest value as often as possible. I've used this subscription pattern in

the multiway demo of the application, described in chapter 3.2.5. In the other demos, I just refreshed and fetched the value often by querying the constraint system for it, simply because of simplicity: it fits well into the `useFrame`-hook detailed later, where I update the constraint system with values each frame. When updating values in this manner, it's easy to get the values as well, contrary to using another pattern to get this done. In this project, it has led to simpler and more compact code as well as faster prototyping. However, a subscription-based approach is recommended for further system expansion.

Details of event types when subscribed to a variable	
Type	When
`{ value: T }`	new value
`{ value: T, pending: boolean }`	new value + pending changes
`{ error: any }`	new error value
`{ error: any, pending: boolean }`	new error value + pending changes
`{ pending: true }`	variable pending

Table 2.1: Details of the different event types when subscribed to a variable. [18]

The subscription-based pattern also allows us to fetch additional data about how the constraint system currently processes the variable. In table 2.1, we can see the details of events we can fetch in a subscription on a variable. This allows us to create extra logic in special cases, such as when a value is pending changes or when something errored. This is very useful, especially when we want to design robust systems.

2.5 JSX

Many of the code snippets presented in this thesis will be either TypeScript-code or JSX-code. TypeScript is a superset of JavaScript, and it has additional features, most notably a type system. In this paper, I will not go into details about TypeScript, as the code is straightforward and can be read without particular knowledge about TypeScript. If you understand pseudo code, you can pretty much intuitively understand TypeScript code.

JSX, on the other hand, might be a bit harder to read. It is a mix of JavaScript (or TypeScript as used in this project) and XML-like tags. This syntax was made popular by the React library. Both the physics and rendering libraries support this syntax, so this syntax will be the design approach when composing scenes and physical worlds.

Here is how React describes JSX: "It is a JavaScript syntax extension popularized by React. Putting JSX markup close to related rendering logic makes React components easy to create, maintain, and delete." [26]

JSX will be used for designing the site layout, assembling the scene, and assembling the physics world.

2.6 Physics

When flying a drone, many parts must work together to make it fly stable. Each propeller's orientation, thrust, and angle work together to make the drone hover and propel it to different points in space. Not everything is implemented from scratch in the demos created - this would be a much larger project. This thesis focuses on using Multiway Dataflow Constraint Systems to power an autopilot but at a higher level than controlling individual propellers. Therefore, the project takes some shortcuts, but no shortcuts that will defer value from the demos and what the MDCS system can provide.

For the drone, the rotation is disabled, as this is not a crucial part of the functionality and does not add anything of value to these specific demos currently implemented. In the demos, we only care about moving the drone between points; rotation is not essential **yet**. After a higher-level functionality has been proven to work, it makes more sense to go deeper into detail. Still, as of now, the focus is on higher-level autopilot functionality. This project would go out of scope to implement forces coming from all four rotors. This would require much more work to get a stable drone and control it. For instance, if we are first doing that, we might calculate forces depending on how the propellers are angled as well and then continue to go deeper into simulation details. In the future, as the demos increase in detail and complexity, this lock should be removed. Although the demo will work with rotations enabled, unexpected behavior can arise because we have used other higher-level functionality. If the rotation lock was disabled and the drone was to crash with another object while hovering, the drone would start spinning around itself but still hover. That is because our force comes from one point and does not follow the rotation of the drone. So if it crashes and hovering mode is active, it will still apply forces upwards, regardless of the rotation of the drone is rotated. Forces from the drone that follows the rotation would not be hard to implement, but it again goes away from the focus of the thesis, as it is not needed yet. Maybe if the project grew, it could be more valuable to test this system at more "low-level" operations, such as hovering using individual force on each propeller.

When it comes to positioning the drone at an altitude, the project uses a delta for the hovering functionality. This is done so the drone does not have to be at an exact altitude. It would be impractical to say the drone should be at exactly $y = 1$, since this is hard to measure in real life due to many possible uncertainties, e.g., sensor accuracy. Instead, we can provide a delta and say we are happy when the y-position of the drone is between 0.95 and 1.05 in the y-value. To hit y=1 in a continuous domain of y-coordinates in real life is practically impossible.

2.6.1 Physics Library

I used a physics library called *react-three-rapier* for this project. This choice was made mainly because it integrates well with another library I used, *react-three-fiber*.

Using *react-three-rapier*, we can assign physical elements using JSX (JavaScript XML) tags. To use this library and create a physics world, we must wrap all

physics elements within the `<Physics>`-tag. Inside here, we can add physical objects. For instance, we can add a rigid body through the `<RigidBody>`-tag and assign it properties such as mass. We can also alter the `<Physics>`-tag, e.g., provide it a custom gravity if we wanted to simulate other planets.

Here is how a simple world with a ground and a drone can look using these tags. Gravity will affect the drone since it is used inside the `<Physics>` tag. The ground is set to be fixed so it is not affected by gravity (or other forces such as collisions) and will not fall.

```
<Physics>
    <Ground size={new Vector3(20, 1, 20)} />

    <Drone
      animate
      debug={false}
      initialPos={new Vector3(0, 0, 0)}
      destinationPostion={new Vector3(0, desiredAltitude, 0)}
      modes={[Mode.Hover]}
      robotModeActive={true}
    />
</Physics>
```

The library makes it simple to apply forces to a rigid body, and it can be done in a single method call: `robot.applyImpulse(forceVector, true)`. The last parameter in the function call decides if the rigid body should be awoken. This method call will be the main source of applying forces in the scenes and will be what pushes the drone around in the world. In some of the implemented systems that use constraint systems, some of the methods will calculate a force vector. This force vector is then applied later using `.applyImpulse(...)`.

The physics library also provides an essential function for hooking into the physics loop, which is `useFrame`. This function allows us to provide a function it should run for each frame. This will be useful for reasons explained later, but one of the things it is used for is to check if the drone is out of bounds of the scene, and if it is, re-spawn it such the user can see it.

2.6.2 Sensors

The drone should be able to observe its surroundings like real drones do. The drone should not depend on the user to give the coordinates of all the other objects in the scene but rather observe the environment by itself. This way, the drone can be put into the world in an independent manner.

For the drone to observe its environment, the project simulates sensors, specifically distance sensors. There are more ways to observe the environment, e.g., through a camera, but that would require significantly more work and probably some machine learning as well. The distance lasers were implemented using sensor colliders from the physics library. Those colliders will detect if other rigid bodies are within the sensor's volume. In my project, sensors are implemented as boxes that protrude from the drone and can be seen visually so that the user can see when the drone detects something intersecting the sensor "beam".

When other rigid bodies intersect a sensor, it fires an event, and we can grab a reference to the rigid body that entered the sensor and can then fetch its coordinates. This functionality that *react-three-rapier* provides makes it simple to implement these sensors.

These sensors try to simulate a distance laser such as LiDAR. The reach of the laser on the drone is minimal by choice, so we can observe what happens when it gets closer to an object and have less time to react. The drone can only see four directions: forward, backward, left, and right. This would be contrary to modern systems, as they most likely will have full 360 degrees of "vision", e.g., by using a LiDAR that rotates. The rigid body objects the physics library provides make getting sensor data from the drone easy. Some of the properties we can fetch include position, linear speed, angular speed, mass, etc.

Chapter 3

Design and Implementation

This chapter will first present an overview of how the application is built. This section will contain code that displays how the different parts of the system work together. After this, each demo will be shown with screenshots, along with an explanation of the demo, and their relevant HotDrink code. Then instructions on how to expand upon this project are detailed. After this is a description of the development method, followed by explanations of how the drone was modeled to reflect real-life properties. In the end, a side-by-side view of a traditional- vs a MDCS-approach of a hover function will be shown, and the differences highlighted.

3.1 Architecture

The whole system comprises three main parts: rendering, physics, and custom drone controller code; the autopilot. There is also a minor part of this system, which is the GUI that allows us to change the state of the simulation, which is the functionality brought by the library *Leva*[22]. For an overview of the architecture, see figure 3.1. In this figure, we can see the parts of the system, what the parts are made up of, and how the different components communicate with each other.

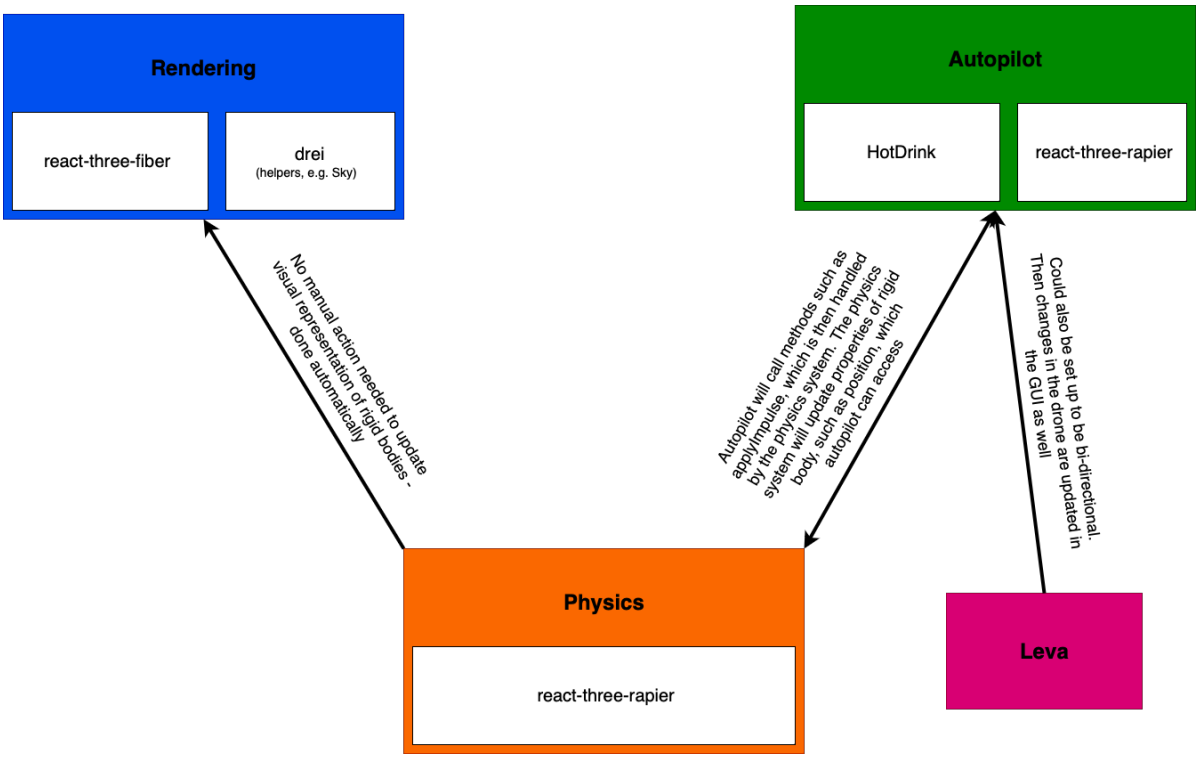

Figure 3.1: Overview of the simulator's components and relations.

3.1.1 Rendering

The rendering component is comprised of *react-three-fiber*, along with *drei*, which provides "useful helpers for react-three-fiber" [21]. The library *react-three-fiber* allows us to express the elements within a scene using JSX syntax 2.5. This makes it easy for us to add new elements to the scene or alter existing ones. We need to provide a canvas context to create a scene that can hold objects to be drawn to the screen. Using JSX, we can add the top-level tag `<Canvas>`. Graphical elements inside this tag can be rendered on the screen. This approach makes prototyping easy, as there is minimal effort to add new elements to the scene compared to writing everything imperatively in TypeScript / JavaScript code.

Now, consider the following example, which is how the project is structured to create a scene with a physics world.

```
<Suspense fallback={null}>
  <Canvas shadows camera={{ position: new Vector3(0, 20, 32) }}>
    <Preload all />
    <Sky sunPosition={sunPosition} turbidity={0} />
    <Stars radius={125} depth={40} count={6000} fade />
    <ambientLight intensity={0.56} />
    <directionalLight
      castShadow
```

```
      position={sunPosition}
      intensity={0.7}
      color="white"
    />
    <directionalLight
      castShadow
      position={sunPosition.clone().setZ(-sunPosition.z)}
      intensity={0.3}
      color="white"
    />
    <OrbitControls />

    {/* Physics world */}
    <Suspense
      fallback={
        <Html center style={{ whiteSpace: "nowrap" }}>
          Loading physics world...
        </Html>
      }
    >
      <PhysicsWorld activeDemo={activeDemo} />
    </Suspense>
  </Canvas>
</Suspense>
```

The example illustrates the simplicity of using this approach with JSX syntax. At the top level in the code, we have a `<Suspense>` tag, which only returns some placeholder information to the user while the `<Canvas>` is loading. This information can, for instance, be a spinner to indicate the canvas is loading. Underneath that tag, we can find our `<Canvas>` tag, and we are giving it some properties. We give it the `shadow` property, which is syntactic sugar for `shadow={true}`, which means to enable shadows for the scene. Then we set the camera position to where we want to see from. We find the elements that set up the scene underneath the `<Canvas>` tag. Some of the tags do not represent any element we can see, such as `<Preload all />`, which is used for the scene to pre-compile. Another tag that is not a graphical element is `<OrbitControls />`. This tag provides controls for the user to see around the scene - this tag is what allows us to move around in the scene using a mouse or a touchpad. It is a convenient way to add controls, enabling us to look around and even move the camera. Some of the elements that can be seen are `<Sky ... />`, which adds a sky to our scene, along with some light provided by `<directionalLight ... />`. The scene includes a `<PhysicsWorld>` tag, which opens up the door to our physics worlds. This will be presented closer in chapter 3.1.2.

Nothing has to be done manually on the developer's side to update the rendering. This is handled automatically by the library *react-three-fiber*. The other rendering library, *drei*-library), provides some valuable components for *react-three-fiber*. When we add physics components to a canvas (under the `<Physics>` tag), the changes in the physics world will automatically reflect in the render. The physics library lives within the same **react-three** ecosystem, playing well together.

3.1.2 Physics

Each component in the physics world should go under the main `<Physics>` tag, which again can be included under the `<Canvas>` element. The `<Physics>` tag allows us to add components that can be affected by the physics world, such as a `<RigidBody>`. `<RigidBody>` represents a body that physics can affect, such as gravity or a collision with another object. For example, if we were to add a simple cuboid rigid body to a scene, we would observe it fall due to gravity.

The physics component of this project, as seen in figure 3.1, communicates with the autopilot through the `<RigidBody>` component, which we can add under a `<Physics>` tag. This rigid body represents the drone, and the custom controller code in the drone will have access to this `<RigidBody>` component. This allows the drone to read physical properties from its physical body: mass, position, rotation, angular velocity, linear velocity, etc. This is data that modern drones also access through their sensors. We can also communicate with the rigid body within this physics system through calling methods such as `applyImpulse(...)`. This implies that we have a two-way communication channel (input and output) between the world and the drone, as we can both get sensor data from it, and we can apply forces to it. `applyImpulse(...)` can be used to apply forces in a particular direction and forms the basis of how we control the drone. It simulates a "push" in a given direction.

The drone's autopilot also gets input from another library called *Leva*, which adds a window where users can edit states. This makes it much easier to debug and fine-tune the state of the drone. Initially, this was another two-way binding, as the autopilot could receive and update the window with its values, such as linear velocity magnitude. However, this was later dropped, as it involved some additional complexity. It was sounder to keep things more straightforward in the project's core to keep the architecture simpler and more accessible for people to expand later.

3.1.3 Drone Controller

The drone controller (the autopilot) is split into different parts; the drone's rigid body, modes, sensors, and input from the mission planner. A mission planner is where you define a mission for the drone. In this thesis, a mission planner means a specification of which modes the drone should run, along with some extra parameters that the user can define (e.g., altitude). In modern mission planners, you can define much more, such as restricted areas or a custom landing spot to go to if something goes wrong during the mission. In the project, we simulate a mission planner through the *Leva*-window (e.g., can set desired height) and through the canvas (e.g., can set a destination point with a mouse pointer).

The drone's rigid body lives in `Drone.tsx`, which is a JSX file with TypeScript support, and represents the drone itself. The top tag returned in the exported function in this file is a `<RigidBody>`-tag. This `<RigidBody>` tag allows the physics system to simulate this object. When starting the demo, one can notice the drone will be affected by gravity and fall to the ground if no modes are activated. In the example below, we are also setting mass to be 1 unit, and we are setting the initial position to be either a position sat by the developer when

placing the drone or a default position. `Vector3` can be treated as a point in this context.

```
<RigidBody
    mass={1}
    ref={rigidBody}
    position={
      props.initialPos ?? new Vector3(0, 10, 15)
    }
    canSleep={false}
  >
```

Modes

Modes are where the MDCS-code live, powered by *HotDrink*. They represent different operations the drone should do, such as hovering. In modern mission planners, a mode could be, for instance, to follow a person using a camera to capture footage of yourself without relying on other people to control the drone for you.

Just as real-life drones can have different modes, the virtual drone also supports this, for instance, a mode for hovering and a mode for approaching a point. The virtual drone currently supports three modes: point approach (fly-to-point), hovering, and object avoidance. I chose these three modes because they are essential for modern autopilots and are amongst the most critical and crucial modes a modern drone should have. Take the drone from demo three as an example.

```
<Drone
    debug
    initialPos={new Vector3(0, 1, 15)}
    destinationPostion={destinationPosition}
    modes={
      avoidanceMode
        ? [Mode.PointSoftApproach, Mode.ObjectAvoidance, Mode.Hover]
        : [Mode.PointSoftApproach, Mode.Hover]
    }
    robotModeActive={robotModeActive}
  />
```

Here we can see that the drone either has three modes (`PointSoftApproach`, `ObjectAvoidance`, and `Hover`) or two modes (`PointSoftApproach`, `Hover`) depending on if `avoidanceMode` is active. This displays the simple architecture of modes in this drone, as it is only a list of modes given to the drone, which can be easily changed depending on some value. `avoidanceMode` is a variable the user can set through the *Leva*-window, and just looks like a checkbox. The value is passed to the drone, deciding if the modes should be run. This makes it easy for the user to see quickly see how the drone reacts when it activates or deactivates its modes.

3.2 Demonstration

Figure 1.3 shows a screenshot of the entire project page. It consists of three main components on the demo page: a scrollable sidebar to choose which demo to start, the main window where the simulator gets rendered, and an upper right window to alter values inside the simulator, such as the desired drone altitude. In this chapter, all the demos are presented and explained regarding their HotDrink-code. The screenshots presented are only of the main window.

3.2.1 Demo 1

In the first demo, the objective is straightforward. Create a world with gravity, and implement hover functionality in the drone. The user should be able to specify the desired altitude in the *Leva*-window, and the drone should try to hover within a small delta of that value. The user is presented with a canvas containing a fixed ground and a drone that can move around freely. In the upper right corner, the user is free to change the desired height of the drone. The screenshot can be seen in figure 3.2.

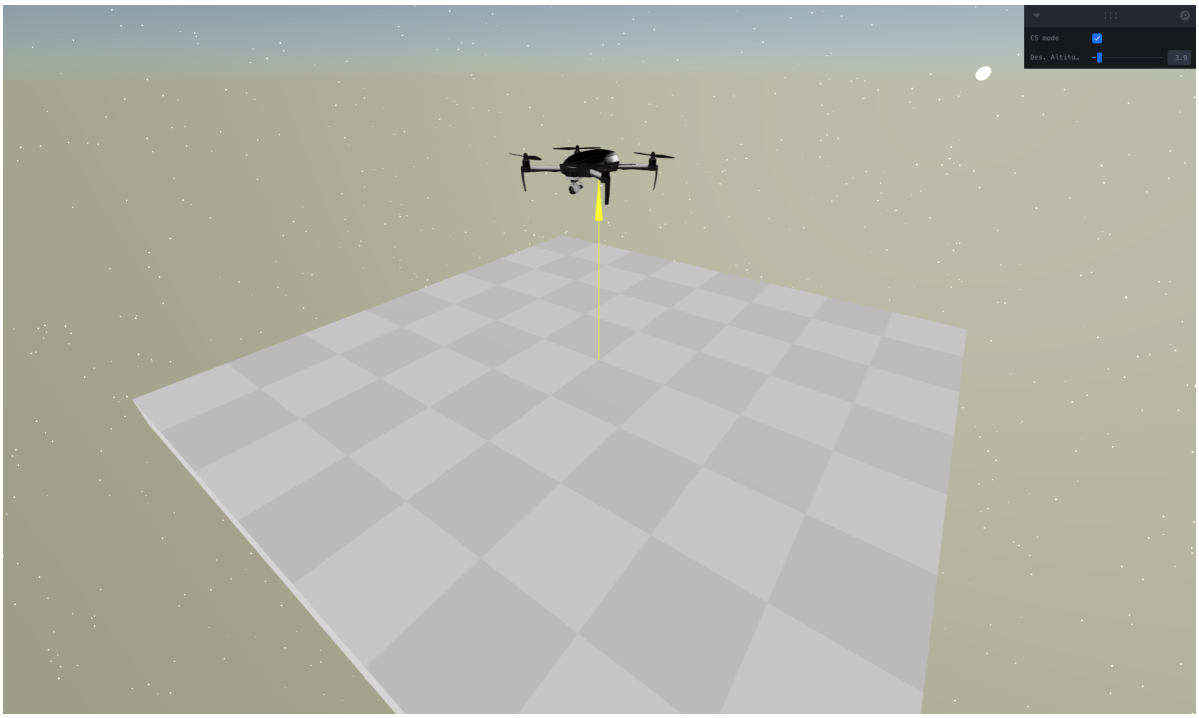

Figure 3.2: Demo 1: a drone using its hover mode

The hovering functionality was implemented using *HotDrink*, and the main variable (the force vector) is computed as shown below. The drone will retrieve this force vector often (requested in each frame) and then apply it. The algorithm is not very sophisticated, but it works well enough to hover the drone around a specified altitude for this purpose. `withinDelta` and `belowDelta` are two other

HotDrink-variables that are computed in the constraint system. The method object is from the *HotDrink*-library.

```
const computeForceVector = new Method(
    4,
    [0, 1, 2],
    [3],
    [maskNone, maskNone, maskNone],

    (withinDelta: boolean, belowDelta: boolean, yVel: number): Vector3 => {
      const unitForce = new Vector3(0, 15, 0);
      const speedTooHigh = yVel > consts.yVelLimit;
      const speedTooLow = yVel < -consts.yVelLimit;

      if (speedTooHigh || speedTooLow) {
        return unitForce.multiplyScalar(-1.5 * yVel); // decelerate
      }

      if (withinDelta) {
        // we want to try to stay in this delta, try to approach a small yVel
        return Math.abs(yVel) < 0.05
          ? unitForce
          : unitForce.multiplyScalar(-4 * yVel);
      }

      // speed is within limits, but we are not within delta
      return belowDelta
        ? unitForce.multiplyScalar(2)
        : unitForce.multiplyScalar(-1);
    }
);
```

There are three input variables to this method and one output variable `force: Vector3`:

1. `withinDelta: boolean` - Signifies if the drone's altitude is within a specified delta, e.g., 0.1 from the desired y coordinate. This means that if any of the following holds

$$desiredY - 0.1 <= y <= desiredY + 0.1$$

   , the within delta variable is considered true.

2. `belowDelta: boolean` - Signifies if the drone's altitude is below a specified delta, e.g., 0.1 from the desired y coordinate. If both this variable and `withinDelta: boolean` equals false, that means that the drone is above delta. If

$$y < desiredY - 0.1$$

   holds, the below delta variable is considered true.

3. `yVel: number` - The velocity of the drone in the y-direction. We need this to adjust our force output if our speed is too high or too low. We also use this value to decelerate.

4. The output of this method is a `force: Vector3`. The drone will use this value for each frame to apply forces that adhere to our mode's goal.

Below we can see the entire process of building the hover system for the drone. This single function will return a function to wrap all of its functionality modularly. The drone will use the returned function to hover. The drone does not need to pass in any arguments when calling the function. The robots should call this function as often as possible to ensure the constraint system knows the latest properties of the drone (e.g., `y` and `yVel`). This code snippet does not show the construction of the variables, methods, and constraints, but this will be detailed later. This code snippet rather shows the general process of creating a mode and what is returned to the drone.

```
export function createHoverSystem(props: Props): () => void {
  const cs = new ConstraintSystem();
  const comp = new Component("HoverMode");

  const consts: Constants = {
    delta: 0.1,
    yVelLimit: 10,
    yDesired: props.desiredAltitude,
    robot: props.robot,
  };

  // HotDrink variables - passing in robot for initial values to variables
  const variables = setupVariables(comp, consts.robot);

  // HotDrink methods
  const methods = setupMethods(consts);

  // Adding HotDrink Constraints to component
  addConstraintsToComponent(comp, variables, methods);
  cs.addComponent(comp);

  // This should be called every frame
  return () => {
    // Update constraint system variables
    updateY(variables.y, consts.robot);
    updateYVel(variables.yVel, consts.robot);

    // Apply force from constraint system variable
    const forceVector = getForceFromConstraintSystem(variables.force);
    consts.robot.applyImpulse(forceVector, true);
  };
}
```

3.2.2 Demo 2

In the second demo implemented, the goal was to make a drone navigate towards a goal but slow down as it gets closer. The MDCS system was responsible for slowing down the max speed of the drone as it got closer to the point. The idea

was that for fragile destinations, such as a wind turbine, you would want to slow down when approaching it and stop going any closer when getting close enough. In this case, the destination goal for the drone was a green sphere (now changed to red to be consistent across demos), whose position can be set by a user. The drone also hovers in this demo and combines its soft approach mode with the hover mode to create the functionality displayed, which shows the modularity of the modes. This mode applies force towards the point, although this is done outside MDCS. However, it is still in the mode logic returned to the drone. It's a ubiquitous functionality in modern mission planners, where users can define a point, and the drone will navigate toward it. In this demo, you may notice an extra option for object avoidance, "Avoidance," which will be further explained in demo 3. The screenshot of this demo can be seen in figure 3.3.

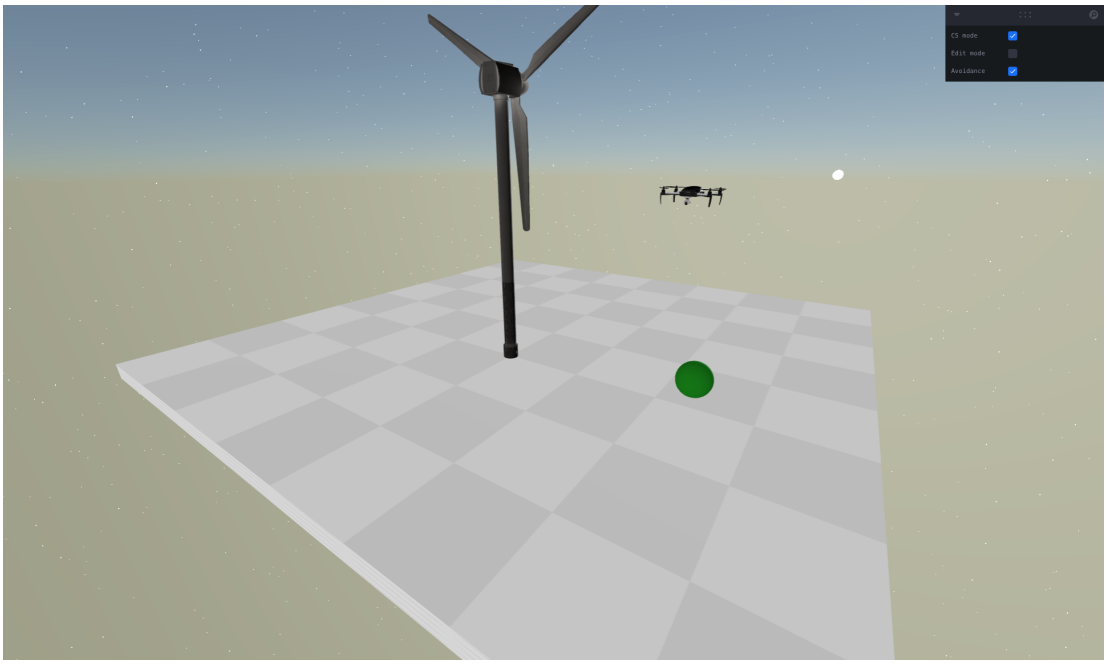

Figure 3.3: Demo 2: a drone approaching the green sphere.

In order to implement this functionality, the following two methods decide the force vector to give to the drone. One of the methods is responsible for calculating the distance in the XZ-plane between the robot position and the destination, which updates the distance variable. When the distance variable is updated, the other method is fired to ensure the constraint is upheld, which again updates the max speed available to the drone.

```
// robotPosition, destination => distance
const computeDistanceToDestination = new Method(
  3,
  [0, 1],
  [2],
  [maskNone, maskNone],
```

```
  (robotPosition: Vector3, destination: Vector3): number => {
    // only want to check distance in 2D-plane (ground)
    const robPos = new Vector2(robotPosition.x, robotPosition.z);
    const destPos = new Vector2(destination.x, destination.z);
    return robPos.distanceTo(destPos);
  }
);

// distance => maxSpeed
const computeMaxSpeed = new Method(
  2,
  [0],
  [1],
  [maskNone],
  (distance: number): number => {
    if (distance < 6) return 0;

    return Math.min(10, Math.log(distance) * 2);
  }
);
```

3.2.3 Demo 3

In the third demo, a more exciting scene appears. Pictured in 3.4, the scene contains a drone, ground, a destination point (red sphere) that the user can place, and a large yellow cube. This cube acts as an obstacle for the drone and is detectable by the sensors on the drone. The yellow cuboid will traverse between the two yellow spheres, making it harder to avoid for the drone. When the drone's sensors detect the yellow cube, it will use its MDCS-system to calculate the drone forces to propel away from the cube. The drone will still try to approach the destination point, but a force vector in the opposite direction of the sensor that detected the cube will be executed (if the cube is within sensor reach). The goal of this demo is again for the drone to reach the destination set by the user. The difference is that object avoidance mode is enabled, so the drone will also try to avoid the patrolling cube using its distance sensors. In real life, the drone would have much more reach with its sensors and maybe have a 360 degrees distance sensor - in this case, we only use four and can easily observe what happens when the drone "sees" an object. The distance sensors are visible to the user as purple cuboid protrusions from the drone. The screenshot can be seen in figure 3.4.

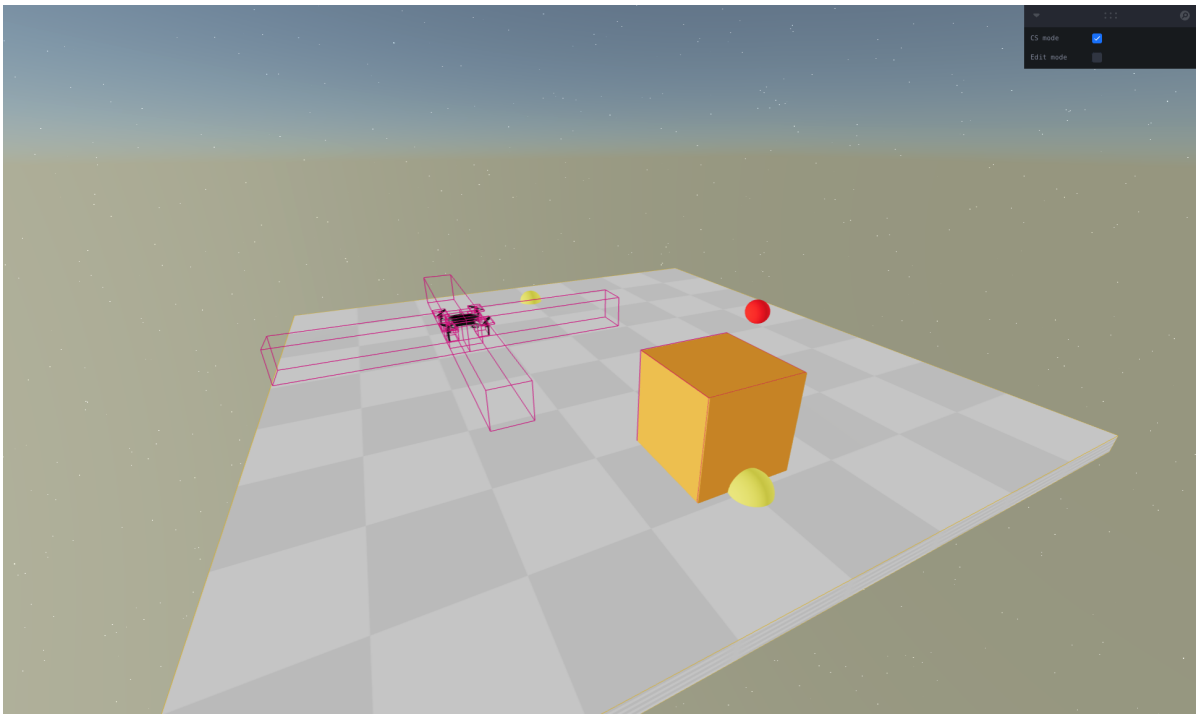

Figure 3.4: Demo 3: a drone avoiding a cube while approaching the red sphere.

The algorithm for object avoidance is somewhat primitive, as we only return a 2D vector in the opposite direction of the sensor that observed the object. However, the algorithm is not the point of interest here; we only care to observe how this system would be laid out. The algorithm in the method can easily be improved.

3.2.4 Demo 4

The fourth demo contains the same drone functionality as demo three, but it has a more complex scene layout to test the drone's avoidance system more extensively. More patrolling cubes are added, and the ground is bigger, so the drone can get a more complex mission. The screenshot can be seen in figure 3.5.

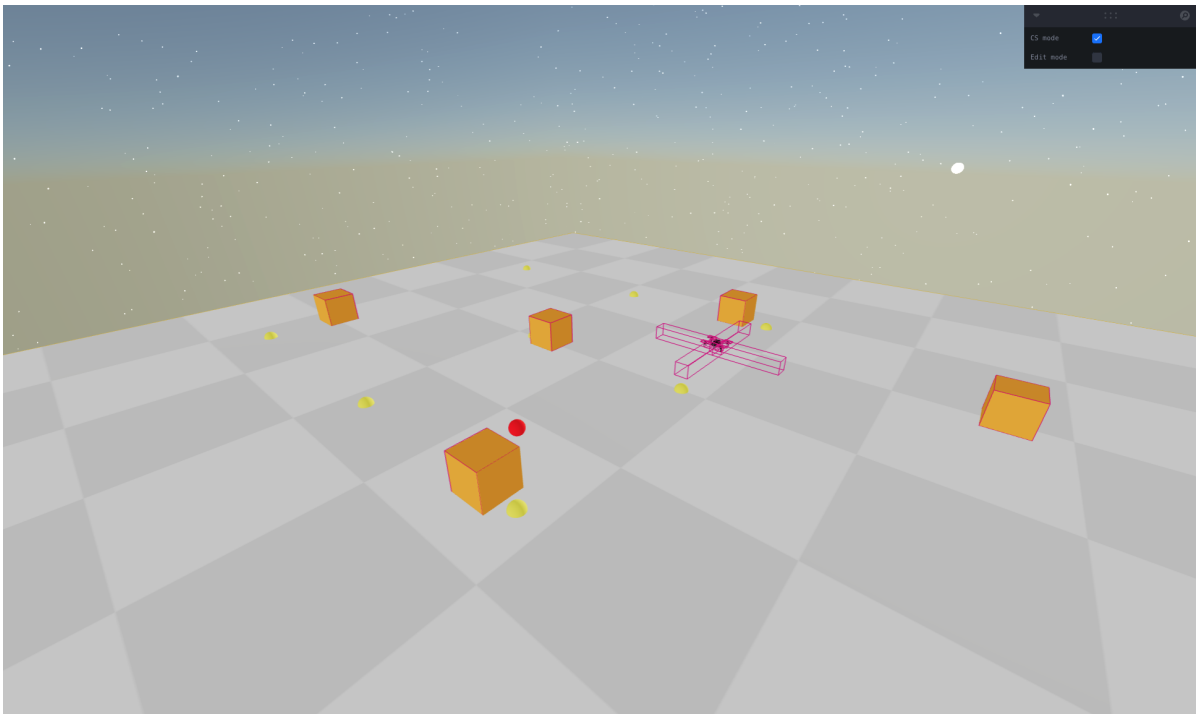

Figure 3.5: Demo 4: a drone avoiding the cubes while approaching the red sphere.

This is the last demo in the simulator. Another demo is implemented using HTML input fields to display how a constraint with multiple methods could work. The demo is still set in a physics context, using points and a vector in 3D space.

3.2.5 Demo Multiway

There was no focus on implementing the simulator demos to use multiway, for reasons explained in chapter 2.3. Since none of the drone demos used multiway flow between data, a separate demo was implemented using number inputs to demonstrate the usage. I used an example that applies to a physics context. Multiway means that a constraint has multiple methods and that different variables in the constraint can update each other, e.g., in the constraint $A = B - 1$, knowing either A or B can update the other value.

In the demo, there are two 3D points, called A and B, and a vector from A to B, called AB. Each of these has an x, y, and z component.

$$A = (x_1, y_1, z_1)$$
$$B = (x_2, y_2, z_2)$$

Then we define the vector from A to B:

$$\vec{AB} = (x_2 - x_1, y_2 - y_1, z_2 - z_1)$$

which all ties together in the constraint for the demo,

$$\vec{AB} = B - A$$

In figure 3.6, an overview of the demo can be seen. At the top of the image, we can see the project has a dedicated page for it to separate it from the demos in the simulator. The demo consists of nine numeric input fields. We have three variables, and each of the variables has three input fields, one for each of their components. When we alter the value in one of the fields, the constraint system is notified of this and automatically oversees that the constraint holds. The input fields are subscribed to the variables, so when the constraint system updates a variable, it is automatically reflected in the input fields for the user to see.

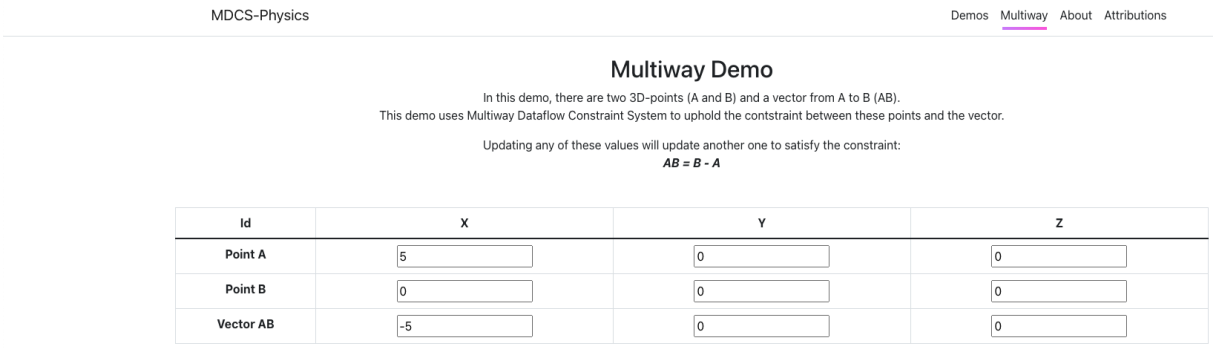

Figure 3.6: A table with nine different inputs representing the vectors and their components.

The code for this demo is a bit different compared to the other demos, as we only utilize a single constraint with multiple methods.

```
const system = new ConstraintSystem();
const component = new Component("Multiway-Demo");

const pointA = component.emplaceVariable("pointA", new Vector3(5, 0, 0));
const pointB = component.emplaceVariable("pointB", new Vector3(0, 0, 0));
const vectorAB = component.emplaceVariable("vectorAB", new Vector3(0, 0, 0));

component.emplaceConstraint(
  "constraint",
  new ConstraintSpec([
```

```
    new Method(
      3,
      [0, 1],
      [2],
      [maskNone, maskNone],
      (pointA: Vector3, pointB: Vector3): Vector3 => pointB.clone().sub(pointA)
    ),
    new Method(
      3,
      [0, 2],
      [1],
      [maskNone, maskNone],
      (pointA: Vector3, vectorAB: Vector3): Vector3 => pointA.clone().add(vectorAB)
    ),
    new Method(
      3,
      [1, 2],
      [0],
      [maskNone, maskNone],
      (pointB: Vector3, vectorAB: Vector3): Vector3 => pointB.clone().sub(vectorAB)
    ),
  ]),
  [pointA, pointB, vectorAB], // vars
  false
);
system.addComponent(component);
```

This demo is put in the context of physics, and these values could represent many different scenarios. For instance, point A could represent a drone's GPS, point B could represent the destination point it should get to, and the vector could represent the shortest path toward the destination point.

3.2.6 Process

This section presents a general outline for creating a constraint system in a physics context. The examples will use TypeScript so that we can observe the types of objects as well.

In order to have objects that can hold the variables, methods, and constraints, we should create a component that can later be added to the constraint system. Adding elements to a component allows us to easily add or remove all of them from the constraint system as a unit.

```
const cs = new ConstraintSystem();
const comp = new Component("HoverMode");
```

After these objects are made, we can add variables to the component. Here we use the `emplaceVariable`-function, which adds the variable to the component with the given name and returns a variable reference we can use. This reference can be used to update the variable's value, subscribe to it, and directly get its value. Except for some explanatory comments, the code snippets presented in

this section are taken from the project code. In this case, we are looking at the hover function.

```
const y: VariableReference<number> = comp.emplaceVariable(
    "y",  // name of the var in in the system
    robot.translation().y  // initial value
);
const withinDelta: VariableReference<boolean> = comp.emplaceVariable(
    "withinDelta",
    false
);
const forceVector: VariableReference<Vector3> = comp.emplaceVariable(
    "force",
    new Vector3()
);
```

When the variables are added, we can add some methods, and then give the methods to our constraints. Here we use the `emplaceConstraint`-function, which directly inserts a constraint to the component with the given variables and methods. In this case, we are adding one constraint between the y-coordinate of our drone and if it is considered within a delta.

```
const computeWithinDelta = new Method(
    2,          // number of variables used in method
    [0],        // position of input variables
    [1],        // position of output variables
    [maskNone], // specifies if input variables are promises (lazy evaluation) or not

    // the function containing the code to satisfy the constraint
    // in this case, the output would be the withinDelta-variable,
    // as seen furter below
    (y: number): boolean => {
      return (
        consts.yDesired - consts.delta <= y &&
        y <= consts.yDesired + consts.delta
      );
    }
);

...

// Here we are adding a constraint to the component
comp.emplaceConstraint(
    "withinDeltaConstraint",  // name of the constraint
    new ConstraintSpec([methods.computeWithinDelta]),  // providing the method(s)
    [vars.y, vars.withinDelta],  // providing the variables
    false  // if constraint is optional
);
```

Observe that the last argument given to the `emplaceConstraint`-function is `false`. This denotes if the constraint is optional. There are two kinds of constraints the system can use. “The constraints in the system are partitioned to

must and optional constraints; the latter are totally ordered according to their strength (we also use the term priority, and talk about the priority order)."[8]

All that remains is to add our component to the system, and then we can update it.

```
cs.addComponent(comp);

// Updating was not needed in my case for the drone demos
// as I updated the variables directly. This causes
// the system to update to satisfy the constraints
cs.update();
```

I will use more examples from the code in 3, but the code presented shows the main ideas applied when designing a system using this process. There are other ways to do this, such as using the domain-specific language provided by *HotDrink*, but I did not use this for reasons presented in chapter 5.7.

3.3 Expansion

This thesis focuses mainly on drones, but different types of Robotic Autonomous Systems, worlds, and modes could be added to this project. This section will explain how this could be achieved within this system.

3.3.1 Adding new robots

There are a few things to consider when adding new types of robots. First, if you plan to use *HotDrink*-powered modes, you should add sensors to your new vehicle. This should be used by the modes so they can only use the provided sensors to achieve their goals. If a mode needs more sensors than the robot can provide, then the mode should not start. Since this is a simulator, this is by all means not necessary, as you technically could start the mode anyway. However, such checks add to the realism and force the developer to reason about which sensors the mode should require and which sensors the robot should have. A stricter version of this check could be wise for more realism in the future. Per now, the sensor check goes like this for each mode trying to start:

```
const hasRequiredSensors = checkSensors(
  PointSoftApproachMode.requiredSensors,
  robotSensors
);
if (!hasRequiredSensors) {
  throw new Error(
    "Robot does not have enough sensors to use mode 'PointSoftApproach'."
  );
}
```

The `checkSensors`-function is simple; all it does is traverse the list of sensors in the robot and check them against the required sensors of the mode, as seen below.

```
function checkSensors(
  requiredSensors: Sensor[],
  robotSensors: Sensor[]
): boolean {
  requiredSensors.forEach((element: Sensor) => {
    const robotHasSensor = robotSensors.includes(element);

    if (!robotHasSensor) return false;
  });
  return true;
}
```

You can see that the required and robot sensors are an array containing a type `Sensor`. This is an enum, and it holds all the different available sensors that robots may have. This enum could be expanded for later work, but currently holds these sensors:

```
export enum Sensor {
  // Distance: distance to closest object within that sensor volume.
  DistanceLeft,
  DistanceRight,
  DistanceBack,
  DistanceFront,
  GPS,
  Gyroscope,
  Altitude,
}
```

An array of this enum should be kept in each robot. The available sensors in each robot and the required sensors in each mode should ideally be in a "contract" - specified in an interface. There is no strict standard signature that every robot or mode needs to adhere to at the moment, but this should be implemented. Per now, the drone contains an array of sensors, and it is configured like below. A similar implementation is done for each mode, where they contain a list of sensors they need to work.

```
const availableSensors: Sensor[] = [
  Sensor.GPS,
  Sensor.Gyroscope,
  Sensor.Altitude,

  // "LiDAR distance sensors"
  Sensor.DistanceBack,
  Sensor.DistanceFront,
  Sensor.DistanceLeft,
  Sensor.DistanceRight,
];
```

Now that we've reviewed available robot sensors and the required sensors for the modes, we can create a mode. To actually use the modes desired, we need to call a function `createModes` and pass it the required arguments.

```
const modeFunctions = createModes(
```

```
  props.modes,  // the modes we want to use
  rigidBody.current,  // the reference to the rigid body of the robot
  availableSensors,  // the available sensors of the robot
  props.destinationPostion  // optional: used for modes that require a destination
);
```

This function returns a list of functions the robot should call each frame, one function for each mode created. The variables with `props.` in front of them signify that they were passed to our robot as properties from the parent. In this case, we can consider the parent our mission planner. The mission planner can decide which modes to use and can decide a destination position we want our robot to approach.

When we have retrieved the list of functions, we need to call them each frame. This is done in the function provided to the `useFrame`-function. In the drone file, the process of using `useFrame` to call all of the mode functions returned looks like the code below.

```
useFrame(() => {
    if (rigidBody === null || rigidBody.current === null) return;

    if (props.robotModeActive !== undefined && props.robotModeActive) {
      modeFunctions.forEach((func) => {
        func();
      });
    }

    outOfBoundsHandler(rigidBody.current);
    ...
});
```

Note that for each mode function, we can call it with empty arguments `func()` - this is an excellent example of how it can aid in separation of concerns. The function will use the reference to the robot (given when first created) and update the constraint systems variables. When the variables are updated, the constraints are invalidated, and the constraint system re-computes the variables.

To add another robot, you should also ensure to return a `<RigidBody ... />` tag, as they are critical to apply physics to our robot. To access our rigid body object, we can pass a `ref={robotReference}` property to this tag. In the drone, it looks like this:

```
<RigidBody
    mass={1}
    ref={rigidBody}
    position={
      props.initialPos ?? new Vector3(0, 10, 15)
    }
    canSleep={false}
    >
    ... (colliders and drone model)
```

Any state the drone holds is implemented using React's `useState`-hook. This

is because of re-rendering: if the drone is re-rendered, and a value is just stored in a plain variable, this value will be reset. This does not happen to a state, as it retains its value across renders. This state could, for instance, be kilometers flown and would look like this:

```
const [kilometersFlown, setKilometersFlown] = useState(0);
```

We are returned with a value for the state `kilometersFlown`, and means to update the value through the function `setKilometersFlown`.

3.3.2 Adding new mode

Modes are written in pure TypeScript (or JavaScript if preferred) and do not require any JSX tags. To add a new mode, we first need to add the new mode to `modes.tsx`, which exports an enum over available modes. As seen below, we currently have three of them.

```
export enum Mode {
  PointSoftApproach,
  ObjectAvoidance,
  Hover,
}
```

After adding our new mode to this enum, we must create our mode function, preferably in a separate file. For instance, if we wanted to add a mode for the robot to follow another vehicle, we could create a file `FollowMode.tsx`. In this file, we should also export the sensors we need to operate this mode.

```
export const requiredSensors: Sensor[] = [
  Sensor.GPS,
  Sensor.Altitude,
  Sensor.Camera,  // to detect other vehicle
  ...
];
```

After adding the sensors, we need to create the logic for the mode. This could be done using MDCS as seen previously or by using another approach - as long as it is wrapped in a single return function, it is OK. After the logic for the mode has been integrated as well within the file, the mode should be ready to use. We only need to return it from the main mode function, `createModes`. To illustrate how this looks, this is how it is done for the object avoidance mode:

```
case Mode.ObjectAvoidance: {
    const hasRequiredSensors = checkSensors(
      ObjectAvoidanceMode.requiredSensors,
      robotSensors
    );
    if (!hasRequiredSensors) {
      throw new Error(
        "Robot does not have enough sensors to use mode 'ObjectAvoidance'."
      );
    }
    modeFunctions.push(
```

```
      ObjectAvoidanceMode.createObjectAvoidance2DSystem(robot)
    );
    break;
  }
```

Following these steps, a new mode can be added and can then be used for any robot, given the sensors are fulfilling the required sensors from the mode.

3.3.3 Adding new world

Per now, each demo has its unique world with a unique environment. The world can consist of physical objects, such as a ground, cube, spheres, or custom rigid bodies. Developers can easily create their own world with their own environment. To add a new world, you first have to interact with the React system and figure out what is the current active demo. That is passed down to `PhysicsWorld.tsx`, which gets the current active demo number. From there, we can load our world lazily (we don't need to load all demos simultaneously). We also pass a key to each world, giving it a random UUID. This is to let React know it can create an entirely new instance of the world and not save any states each time we load the world. We want a new world each time we load it up, giving it a unique key. The code for selecting the world looks like this:

```
import { lazy } from "react";

const World1 = lazy(async () => await import("./worlds/World1"));
const World2 = lazy(async () => await import("./worlds/World2"));
const World3 = lazy(async () => await import("./worlds/World3"));
const World4 = lazy(async () => await import("./worlds/World4"));

interface Props {
  activeDemo: number;
}

/* We want to reset the states of the physics worlds on demo change. */
export default function PhysicsWorld(props: Props): JSX.Element {
  if (props.activeDemo === 1) {
    return <World1 key={crypto.randomUUID()} />;
  } else if (props.activeDemo === 2) {
    return <World2 key={crypto.randomUUID()} />;
  } else if (props.activeDemo === 3) {
    return <World3 key={crypto.randomUUID()} />;
  } else if (props.activeDemo === 4) {
    return <World4 key={crypto.randomUUID()} />;
  } else {
    return (
      <p className="text-center h2 mt-5 text-danger">
        Could not find demo with id={props.activeDemo}
      </p>
    );
  }
}
```

To add a world, one could add the created world to the if / else if / else clause and give it a random key. Each world, e.g., `<World1 ... />` consists of JSX tags describing the physics environment. This means that inside the worlds, they return a `<Physics>` tag at the top level. Here we can see how this looks in `<World1 ... />`:

```
export default function World1(): JSX.Element {
  const [{ desiredAltitude }] = useControls(() => ({
    desiredAltitude: {
      value: 10,
      min: 3,
      max: 15,
      label: "Des. Altitude",
    },
  }));

  return (
    <Physics>
      <Ground size={new Vector3(20, 1, 20)} />

      <Drone
        animate
        debug={false}
        initialPos={new Vector3(0, 0, 0)}
        destinationPostion={new Vector3(0, desiredAltitude, 0)}
        modes={[Mode.Hover]}
        robotModeActive={true}
      />
      <arrowHelper
        args={[new Vector3(0, 1, 0), new Vector3(0, 0, 0), desiredAltitude + 2]}
      />
    </Physics>
  );
}
```

We observe that we utilize a hook called `useControls` - this enables us to retrieve information from the *Leva*-window. Below this, in the return statement, we see that we return a `<Physics>` tag, with some elements inside, notably the `<Drone ... />` tag, and pass it the property `desiredAltitude`, so that the drone can hover to that altitude. With this in place, we can now add the demo to the sidebar along with some logic for setting our new demo to an id corresponding to its position in the sidebar.

3.4 Development method

The approach for this application was to create a simulator for using *HotDrink* in a physics context and then add different demos to showcase the usage. The goal was to start with simpler demos and then slowly build complexity in the systems until a fairly complex demo is built, and then evaluate how the *HotDrink*-powered systems work. A complex enough demo was not implemented

to showcase the full potential and advantages/disadvantages of using MDCS. However, four demos were created, and they all show how essential features of an autopilot could be powered by the *HotDrink* library. The demos were useful for showing how MDCS could be fitted to a robotic system.

The first demo took the most time to create, as the technology stack had to be set up along with an architecture that made sense. After this, the development of new demos went faster. Initially, the focus was more fine-grained. The initial process was to simulate a drone as closely as possible. I implemented the balancing of the drone's rotation in the Unity game engine by using torque, and the plan was to simulate this with force from the four rotors at a later point. However, the focus shifted from implementing low-level functionality to focusing on an autopilot's higher-level functionality, which could lead to more exciting and interesting features in less time.

This system ties around React, and much of the work is done in a React-like way. This is why many of the core libraries such as *react-three-rapier*, *react-three-fiber*, *drei*, and *Leva* are used with JSX-tags. Using React was chosen because I knew it had a lot of help available in the React communities, and I wanted to explore React in itself. In the project's beginning phases, *Leva* helped tremendously with fine-tuning the parameters given to the *HotDrink*-system. It was used more in the project's beginning phases but was less extensively used in the end due to architectural changes. Now it currently only allows the end-user to choose some values and see how the autopilot reacts to them. The autopilot does not change the GUI.

Below is the final software stack of the project.

- React [25] - for creating the view/layout of the website
- HotDrink [15] - the library providing an implementation of MDCS
- TypeScript & JavaScript - used mainly TypeScript, used for adding functionality
- Leva [22] - used for quickly changing state values in a visual manner
- React three fiber
  - Acts as a wrapper around three.js [30], used to layout the canvas
- React three rapier [27]
  - Acts as a wrapper around the rapier physics engine [24], used to create the physics world
- drei - provides useful elements for the canvas

3.5 Code structure

3.5.1 Connectivity with HotDrink-Powered Systems

Maybe the most notable piece of the architecture is how the drone calls the systems created using *HotDrink*. In the first approach, the drone would have the *HotDrink* system directly on itself. The drone would then relay its rigid body

values to the system and use the values computed by the constraint system to run the current mode. This, however, had its downsides. It makes the mode tightly coupled with the robot and does not work well with separation of concerns. It also leads to a more messy structure in the code. A drone was made per mode in the beginning phases, as the mode and robot were so tightly coupled, but this was quickly discarded.

The final approach is much more modular and cleaner than the first one. Still, it might be a bit overly complex and could be refactored to be simpler to use. The current approach is to make a mode return a single function without arguments to the drone. This function should be called in each frame in the drone (in real life, it should be called as often as possible to get more accurate results).

3.5.2 HotDrink Methods and Variables

Implementations for the methods differ; some need more complex logic, while some can naturally be expressed with an equation. For example, computing the max speed of the drone relative to the distance to the wind turbine (it should slow down when getting close) could be defined the following way:

$$maxSpeed(distance) = \begin{cases} 0 & \text{, if } distance <= 1 \\ log_{10}(distance) & \text{, if } distance > 1 \end{cases}$$

Figure 3.7 plots this function, representing the constraint between the distance and the max speed. As an extra check, one could define an upper value bound of the max speed (`maxSpeed = (maxSpeed > 20) ? 20: maxSpeed`). In this case, it's okay, as `log10` climbs in value really slowly. In fact, when using the distance in meters from the earth to the sun, the max speed would only be approx 11 m/s. In this case, I define max speed as we want to only apply acceleration up to this max speed but not apply acceleration when above this threshold. This concept was used for the point approach mode, where we apply less force the close we come to our destination.

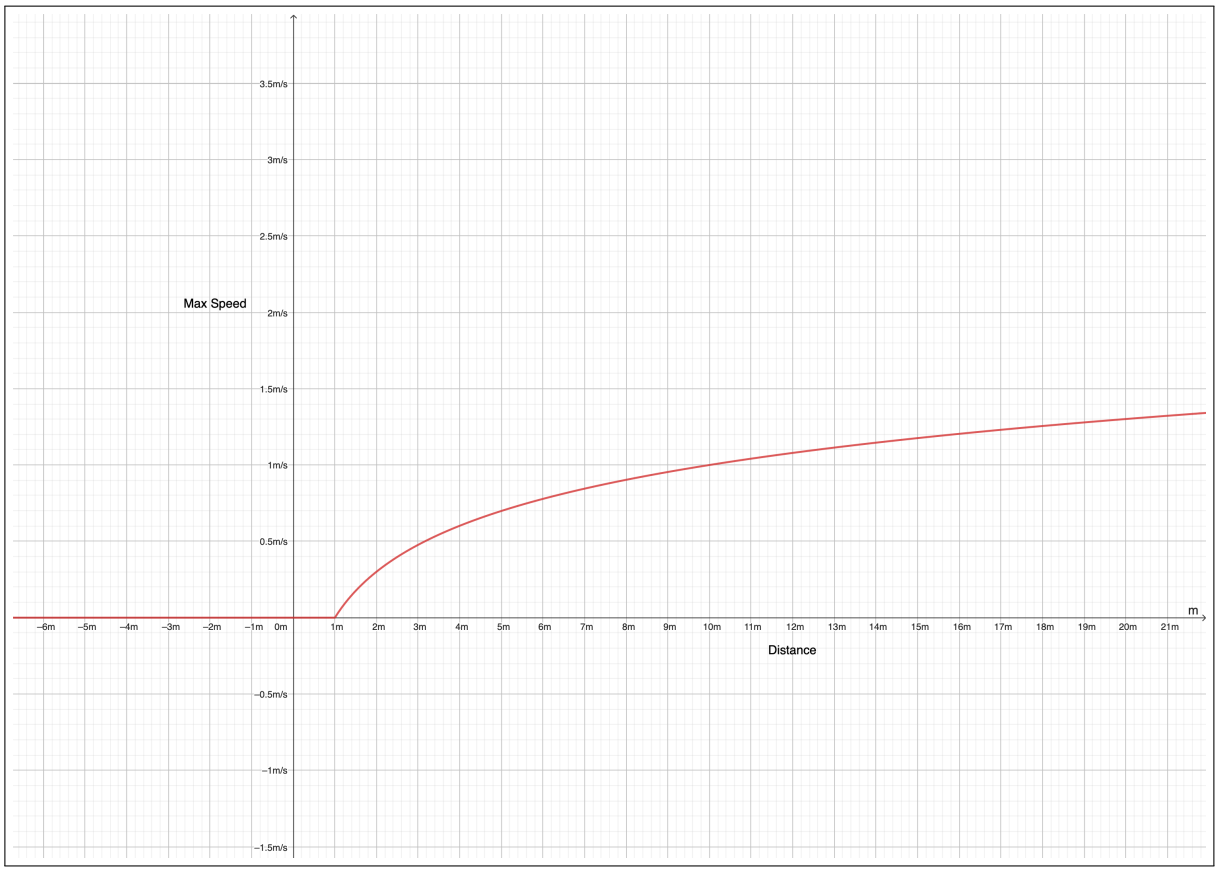

Figure 3.7: An graph showing an example of a max speed constraint.

Some of the HotDrink variables are only input variables and will not be changed by the constraint system itself. Examples of this are the coordinates of the robot and the speed - these *HotDrink*-variables are updated directly through the variable reference. On the other hand, some variables are the opposite - they will not be updated directly but only computed by the constraint system. An example of this is computing whether the drone's altitude is less than the current altitude of the drone minus a delta. Provided with the current altitude and the delta, the constraint system will compute the `belowDelta` value.

3.6 Modeling the Real World

One of the main questions considered in this thesis was how to combine MDCS, the robot, the scene, and the world intuitively and logically, such that their relations make sense. Multiple approaches were tested.

One of the most notable features to include was letting the drone observe the world to avoid hitting objects. The first approach to implement this consisted of adding sensor boxes in the world (not connected to the drone itself), and if something intersected with those sensors, notify the robot of this. But this is

not how it works in real life; the world does not "tell" a robot which objects are located in a particular area; the drone must observe this for itself. Therefore, this approach was deemed erroneous and not valid to real-life approaches.

The second approach was to give a reference of a rigid body object, such as a wind turbine, so that the drone could see its GPS coordinates. While this is more applicable to the real world, where we can tell the drone the position of other objects, it's still not the correct approach for this solution. We want the drone to be completely autonomous and, in principle, no need to rely on the user to tell it where objects are. The robot should rely on its internal sensors to implement this feature.

The third and final approach consisted of adding sensor boxes that protrude from the drone to act as distance sensors. These boxes can detect if something is inside them and retrieve a reference to the object within. While we can get much information from this object reference, we only take the position from the object that intersected the sensor box and used this to calculate the distance to the object. This distance is the only thing from this observation that is given to the object avoidance system, along with which the sensor observed it (front, left, right, or back). This way, we are not "cheating" by getting the information we usually could not retrieve from, e.g., a distance laser in real life.

In the final state of the drone, the `<Drone>` tag consists of three main parts returned: the rigid body of the drone, the sensors, and the model of the drone. The drone component differs a lot from the initial tests with creating robots.

One of the first approaches looked like this when there was still one drone per mode:

```
<RobotPointApproacherAndAvoid
    objectToAvoid={windTurbine}
    destinationPoint={pointA}
    />
```

This is a bad model for a robot, as it should be able to have different modes as long as it has enough sensors and not just one by default. This is fixed in the new approach. In the new approach, we can compose different modes together.

```
<Drone
  ...
  destinationPostion={destionationPosition}
  mode={[Mode.PointSoftApproach, Mode.ObjectAvoidance, Mode.Hover]}
  ...
  />
```

This component is much more flexible and allows for multiple modes to be given.

3.7 Traditional- and HotDrink-implementation

In this section, we will look at two different ways of programming a hover function for the drone. A more thorough comparison between the two approaches is done in chapter 5.7 - this chapter merely presents the different approaches. The code is modified a bit here to keep it more concise and clear. We are not

worried about drone rotations, and we can assume the rotation of the drone is locked. We only consider applying enough force upwards to maintain a desired height, denoted by `yDesired: number`.

Below, this hover functionality is implemented using a *HotDrink*-powered approach. It is a lot of code and could easily be written in much fewer lines (e.g., could use the domain-specific language of *HotDrink*), but this way, everything is typed out with its purpose, and it is easier to see how elements are created and composed. The functionality of this code boils down to three things: assemble the variables and methods for the constraint system, add these variables and methods to the constraint system, and finally return the function that the drone should run every frame. Some repetitive code is omitted from this approach, to make it a bit shorter for the reader, but the essential parts are included. It is a lot of code presented, but it is important to show the extent of the code.

```
export function createHoverSystem(props: Props): () => void {
  const cs = new ConstraintSystem();
  const comp = new Component("HoverMode");

  const consts: Constants = {
    delta: 0.1,
    yVelLimit: 10,
    yDesired: props.desiredAltitude,
    robot: props.robot,
  };

  // HotDrink variables - passing in robot for initial values to variables
  const variables = setupVariables(comp, consts.robot);

  // HotDrink methods
  const methods = setupMethods(consts);

  // Adding HotDrink Constraints to component
  addConstraintsToComponent(comp, variables, methods);
  cs.addComponent(comp);

  // This should be called every frame
  return () => {
    // Update constraint system variables
    updateY(variables.y, consts.robot);
    updateYVel(variables.yVel, consts.robot);

    // Apply force from constraint system variable
    const forceVector = getForceFromConstraintSystem(variables.force);
    consts.robot.applyImpulse(forceVector, true);
  };
}

function setupVariables(
  comp: Component,
  robot: RapierRigidBody
```

```
): HotDrinkVariables {
  const y: VariableReference<number> = comp.emplaceVariable(
    "y",
    robot.translation().y
  );

  ...

  return { y, yVel, withinDelta, belowDelta, force };
}


function setupMethods(consts: Constants): HotDrinkMethods {
  const computeWithinDelta = new Method(
    2,
    [0],
    [1],
    [maskNone],

    (y: number): boolean => {
      return (
        consts.yDesired - consts.delta <= y &&
        y <= consts.yDesired + consts.delta
      );
    }
  );

  const computeBelowDelta = new Method(
    2,
    [0],
    [1],
    [maskNone],

    (y: number): boolean => {
      return y < consts.yDesired - consts.delta;
    }
  );

  const computeForceVector = new Method(
    4,
    [0, 1, 2],
    [3],
    [maskNone, maskNone, maskNone],

    (withinDelta: boolean, belowDelta: boolean, yVel: number): Vector3 => {
      const unitForce = new Vector3(0, 20, 0);
      const speedTooHigh = yVel > consts.yVelLimit;
      const speedTooLow = yVel < -consts.yVelLimit;

      if (speedTooHigh || speedTooLow) {
```

```
        return unitForce.multiplyScalar(-1 * yVel); // decelerate
      }

      if (withinDelta) {
        // we want to try to stay in this delta, try to approach a small yVel
        return Math.abs(yVel) < 0.05
          ? unitForce
          : unitForce.multiplyScalar(-1.5 * yVel);
      }

      // speed is within limits, but we are not within delta
      return belowDelta
        ? unitForce.multiplyScalar(2)
        : unitForce.multiplyScalar(-1);
    }
  );

  return {
    computeWithinDelta,
    computeBelowDelta,
    computeForceVector,
  };
}

function addConstraintsToComponent(
  comp: Component,
  vars: HotDrinkVariables,
  methods: HotDrinkMethods
): void {
  comp.emplaceConstraint(
    "withinDeltaConstraint",
    new ConstraintSpec([methods.computeWithinDelta]),
    [vars.y, vars.withinDelta],
    false
  );

  ...
}

function getForceFromConstraintSystem(
  forceVarRef: VariableReference<Vector3>
): Vector3 {
  const forceVec = forceVarRef.value?.value;
  if (forceVec !== null && forceVec !== undefined) {
    return forceVec;
  }
  return new Vector3();
}
```

This results in a lot of code for functionality that is not too complex. Now,

consider a more traditional approach of the MDCS-implementation of the hover.

```
export function hover(robot: RapierRigidBody, yDesired: number): void {
  const y = robot.translation().y;
  const yVel = robot.linvel().y;
  const yVelLimit = 1.1;

  const delta = 0.2; // acceptable offset
  const withinDelta = yDesired - delta <= y && y <= yDesired + delta;
  const belowDelta = y < yDesired - delta;

  const unitForce = new Vector3(0, 20, 0);
  const speedTooHigh = yVel > yVelLimit;
  const speedTooLow = yVel < -yVelLimit;

  if (speedTooHigh || speedTooLow) {
    const force = unitForce.multiplyScalar(-1 * yVel); // decelerate
    robot.applyImpulse(force, true);
    return;
  }

  if (withinDelta) {
    // we want to try to stay in this delta, try to approach a small yVel
    const force =
      Math.abs(yVel) < 0.05 ? unitForce : unitForce.multiplyScalar(-1.5 * yVel);
    robot.applyImpulse(force, true);
    return;
  }

  // speed is within limits, but we are not within delta
  const force = belowDelta
    ? unitForce.multiplyScalar(2)
    : unitForce.multiplyScalar(-1);
  robot.applyImpulse(force, true);
  return;
}
```

This function is much more concise and easier to read than the previous function, but they essentially contain the same functionality. One of the main differences when using the function that follows the traditional approach is that it needs to be called with a rigid body reference each time.

```
useFrame(() => {
    if (rigidBody === null || rigidBody.current === null) return;

    if (props.robotModeActive !== undefined && props.robotModeActive) {
      modes.forEach((mode) => {
        if (mode === Mode.Hover) {
            hover(robot: RapierRigidBody, yDesired: number);
        }
        ...
```

```
      });
    }
});
```

At the core, there is not too much difference between how they are called, even though the traditional approach delivers a reference to the rigid body per call. The traditional approach could also have been laid out, such that it could call a function with no arguments, after first initializing the create-function call with a reference of the rigid body, like the *HotDrink*-approach takes. Therefore, it is important to note that the argument-less function call is **not** a feature brought by *HotDrink*.

Chapter 4

Use cases

Multiway Dataflow Constraint Systems (MDCS) in itself has many use cases and is not limited to only Graphical User Interface (GUI). This thesis is exploring the use cases in physics, specifically in autopilots for systems that maneuver in the physics world. This is already very specific, and for physics in general, MDCS could be applied to many different sub-sections and applications. In the section that the thesis focuses on, Robotic Autonomous Systems, many different types of robots could befit MDCS. It does not have to be exclusively autopilots; radio controller software, traction control for robots with wheels, and route planners are all good candidates. The list goes on, and there is much to be explored further in a physics context using MDCS.

MDCS can be used in all applications where an implementation can be run, e.g., a drone can run the JavaScript/TypeScript version of HotDrink as long as the system has support for running the library. This could be done through Node (`https://nodejs.org/en`), for instance, which allows you to run JavaScript without a browser. An autopilot system could interact with a separate service that can run an implementation of MDCS. An example of this is running Node on a companion computer (a different computer on the same robot that can run more computationally expensive calculations) or connecting to a network service outside of the drone itself. The latter may not be feasible, considering the latency the operations would have - maybe it could function well if there were operations carried out that were not time strict. There is also an implementation of HotDrink in Rust [16]. Rust can be used for embedded devices, meaning a JavaScript implementation of MDCS is not necessarily needed. This is very exciting, and one day we might see a washing machine implemented using *hotdrink-rs*.

When planning to use such a system, one should consider the difference in computing resources needed to run it fast enough. There is some overhead to running *HotDrink*, compared to just running the "traditional" code. This is due to many factors, e.g., the need to create more objects (such as the constraint system object) compared to the traditional approach, which can do the calculations directly and not be abstracted in a *HotDrink*-method. *hotdrink-rs* could maybe aid in this problem, as it is written with a faster language than

JavaScript.

Some other use cases for this system might be in systems that are already in place today and deal with complex relations between nodes but could benefit from *HotDrink*. An example might be an intersection: we have a complex relationship between the colors of the lights, the current amount of cars in each lane, the different directions a car in a lane can take, the time waited, etc. Some systems might benefit from integrating *HotDrink*, and some systems might be better off not using it.

The simulator created can be used to see how a *HotDrink*-powered system works. As well as visually, the code behind the application can give insights into how one might structure the architecture to fit this logic and expand upon it later. Individual elements from this code, such as the drone or the different modes, could also be adopted into other projects. This could provide baseline functionality, which the developer can expand upon later.

Chapter 5

Discussion

5.1 Physics Environment

The physics environment worked well for this project, as no manual computing labor had to be done to visualize the drone's or environment's physical effects. The physics environment allows us to really take complete control over what is happening in the world and add our own effects if desired. Not only can objects be added to the environment, but the environment itself could affect objects, as it is doing with gravity. Some of the following steps to further improve realism could include wind, and it would be exciting to include a hovering mode that accounts for the wind speed and direction.

By using this physics environment, the developer can add visual objects and assign physical properties to them as pleased. The developer should also be careful when doing this, as it is easy to create unrealistic scenes where the physics doesn't add up. E.g., by having fixed bodies (won't move no matter how much force you add to them), unrealistic masses of objects, and acceleration of rigid bodies beyond what the physical counterpart could handle, the developer is introducing unrealistic scenarios. It is essential not to feel like this simulation is always accurate no matter what when it, in fact, can contain errors and unrealistic situations.

5.2 Control

Another question that arose several times is how much control over the rigid body should be given to the MDCS system. It could be given a reference to the rigid body to get complete control over the drone and control it from within the MDCS-methods. Or, it would only be given properties from the drone through variables and then compute a result to a variable that the drone could later use. This project went with the latter approach, as it is more "pure"; it only computes a new variable value and returns it instead of modifying the forces on the drone directly within a method. Some features were also implemented outside the MDCS-system. For instance, in `PointSoftApproachMode`, the MDCS-system is only responsible for calculating the max speed. The logic

for adding forces to push the drone towards the destination point is outside the MDCS-system.

This application still has not used MDCSs full potential. For instance, MDCS many options not considered for this project: priority functionality, optional functionality, activations, promise masks, etc. Future projects regarding this should explore the impact of tweaking with different settings and parameters on the system.

5.3 Simplicity

Although some reading on the HotDrink-library is needed to start using this programming model, it's easy to get started after familiarizing yourself with the key topics. The more intricate part is incorporating it into a project and ensuring the architecture and components are well laid out. The previous iterations of the architecture resulted in code that would not scale well or was overly complex and cumbersome. The architecture in this project is by all means not perfect, but it is scalable enough that the system can continue to be worked on. It took a few iterations to make this work in the desired manner for this project, but now it's modular and structured into logical components. This makes expansion, alterations, and overall changes easier to complete without much overhead. Albeit the system's complexity could be reduced, it has enough modularity for later expansion, and it is easy to add new modes, robots, and worlds.

The way the demos are set up also makes it really easy to create new worlds/environments (or use existing ones) to experiment with robots. Using JSX syntax is one of the keys to making the system modular and easier, and adding whole objects to the scene with potentially only one line makes for efficient and concise programming. We also separate code into logical components, such that separation of concerns is applied, and the developer can shift focus from **how** an object should be added to the scene to **what** should be in the scene.

Even though we are using a digital twin, it is essential to note that it does not mean the digital twin is a hundred percent accurate to the real world. The goal is to get close enough that we can start to form a picture of how MDCS could look in real life. The simulator does not have random events in the project, such as in the real world. In the real world, events that are out of our control can happen, such as unexpected rain or a bird crashing into our drone. Some randomness could be added to our digital twin; however, we don't need it yet, as we only want to test our autopilot in known environments and scenarios.

A digital twin is a useful tool for testing out how something might behave in the real world and can reduce errors and improve the software before starting to use the tool in the real world.

5.4 Determinism

The physics engine used in this project is deterministic, which is important to be able to re-create the experiments and the outcomes. This means that a given

demo will always be in the same state after x number of steps and have the same initial state, which is very useful when we want to re-create scenarios.

The rapier website says: "The WASM/Typescript/JavaScript version of Rapier is fully cross-platform deterministic. This means that running the same simulation (with the same initial conditions) using the same version of Rapier, on two different machines (even with different browsers, operating systems, and processors, will give the exact same results"[10].

For this project, determinism was not so much needed as the general behavior of the drone using MDCS was more important than observing the different scenarios with different autopilot parameters. However, it can become a cornerstone feature if this project is further worked on, or if a particular situation should be re-created.

5.5 Mode Conflict

Since the modes don't check for force vectors returned by other modes, the vectors returned can result in conflict. This happened between the mode where the drone traverses to a point ("PointSoftApproach"), and the object avoidance mode. What happened was that the drone tried to approach a destination point, but on the way, it almost hit a cube. This made the object avoidance compute an avoidance vector that it should use, which was in the opposite direction of the force vector computed by the other mode, "PointSoftApproach". Since the vector from the object avoidance had greater magnitude, the object avoidance still worked slightly but lacked the force to make a big enough impact to avoid obstacles. This was fixed by simply upping the force in the object avoidance mode. In the future, however, a better way should probably be implemented, where the modes have a context for which other modes are running.

5.6 Similarities to React state

React allows us to store the state of a component using the function `useState`. Functions that hook into React and enable you to use different features, such as `useState`, are called a "hook" [6]. We can pass in our desired initial state, which will be the same even though React re-renders the component, flushing all other variables not stored in a state like this. We can use a function returned by this function call to set a new state. As the example below shows, you can use `useState` to store a state, and it returns an array where the first element is the stored state, and the second element is a function to update the state.

```
// 0 is the initial value
const [myNumber, setMyNumber] = useState(0);
```

We can also utilize another hook, `useEffect`, to execute code when the state changes. This function takes in a function to call as a first argument and some dependencies as the second argument. For the examples below, we are adding state values as dependencies. As an oversimplified explanation that serves well enough for this purpose, the function is called if any of the states in the dependency change. As an example, if we wanted to print out the new

value when a state updates, we can do the following:

```
const [myNumber, setMyNumber] = useState(0);

// console.log() is called when using setMyNumber
useEffect(() => {
    console.log("New value", myNumber);
}, [myNumber]);
```

Observe that the last example demonstrates a one-way flow: when the state is updated, a statement to log something out to the console is called. Recall that in the simulator, the multiway-part of MDCS was not used, but rather a one-way approach is implemented with *HotDrink*. This bears similarities to the one-way flow demonstrated here, and we can expand upon this idea with React. Consider the below example, where we first implement multiway flow with React. We say multiway, but at the core, we use two one-way flows to define a multiway flow.

"Planning determines a dataflow. There may be many dataflows that could enforce a system's constraints. ... HotDrink's planner algorithm selects a dataflow that is (intended to be) the least surprising to the user. Slightly simplifying, the planner favors dataflows that preserve the values of variables that the user has changed most recently." [18]

In Reacts case, when any of the dependencies in the example below change, we call the function. Here the constraint $b = a + 1$ is implemented.

```
// b = a + 1
const [a, setA] = useState(1);
const [b, setB] = useState(2);

useEffect(() => { setA(b+1) }, [b]);
useEffect(() => { setB(b+1) }, [a]);
```

This shows how concisely React can define such a constraint. We could also represent a one-way waterfall, where a state update triggers a function call that updates a state and triggers a function call again. Using this approach, we could define the hover function presented earlier using React states and effects to get a flow that is similar to the demos implemented. However, to showcase the multiway capabilities of MDCS, the demo "Multiway" was presented.

5.7 Comparison to the traditional approach

In this section, we are considering the code of the hover function from the code comparison in chapter 3. One of our first observations is that the HotDrink-powered code is much larger. This is due to multiple factors, one of which is that the code is refactored to be very explicit. This is by choice, as the examples presented were to display types and the exact process of building such a system. The code could fit into fewer lines, e.g., using the *HotDrink*-domain-specific language.

We also need to consider the readability of the code and the separation of

concerns. In terms of readability, the traditional approach is the winner. The conventional approach produces code that does exactly what it needs to do, and not more: fewer variables, fewer lines, and less abstraction give the developer an easier experience when figuring out what the different parts of the function do. Adding *HotDrink*-code feels like overkill in such an easy function, as there are no complex or less visible relations between the inputs and outputs of the drone.

The clear winner is the traditional approach for a project of this size and complexity. This is due to more concise code that is easier to read, an equal amount of separation of concerns achievable, and less abstraction for simple operations. However, this highly depends on the developer's choices when developing the system, and **does not necessarily generalize to other systems**. Just using *HotDrink* in this project would probably result in a bit more code anyways, but this is not the sole reason. The project was also developed with a focus on stricter typing, which results in more code. Utilizing *HotDrink* adds more abstraction to the code, which was not needed for this specific purpose. However, for larger systems, this might turn beneficial. The developer needs to evaluate the threshold for when it turns beneficial when designing and reasoning about such a system. In GUI-programming, the threshold for adding *HotDrink* is probably much lower, which means less complexity is needed before adding it becomes beneficial.

5.7.1 Speed

For crucial computations within the autopilot that needs to be computed as fast as possible, such as hovering, using MDCS might be disadvantageous compared to the traditional approach. This is because MDCS will add some overhead for this type of calculation. It might not add much overhead, but for smaller embedded systems, it can be enough to make it disadvantageous. They don't necessarily have that much computing power, and it could be wise to check if the system can afford to change over to use MDCS. However,MDCS can be worth adding if we discover we have sufficient computing power and that MDCS will prove advantageous to the system. There is also the option of using a companion computer on systems that don't have enough power to run MDCS-powered operations on their own.

Chapter 6

Related Work

The original problem description, with having a drone navigate closely around a wind turbine, was inspired by another project called SAMURAI: Sonic Aneometer on a MUltiRotor drone for Atmospheric turbulence Investigation [5]. At the time of coming up with the problem description, the project aimed at creating systems for observing turbulence around a wind turbine, with the goal of researching how it affects the surrounding wind turbines. These measurements could contribute to our understanding of wind turbine wake: a phenomenon where the turbulence and wind disruptions caused by one wind turbine can affect the efficiency of the wind turbine that receives this disrupted wind. Figure 6.1 displays what wind wake looks like.

Figure 6.1: A photo demonstrating wind wake [13].

In figure 6.1, we can clearly see the disruptions that some of the wind turbines have on the wind pattern behind. This wake effect can cause the other wind turbines to lose efficiency, and this is a problem in maximizing energy gathered from such wind turbine farms. Although inspired by the idea of having a drone operate close to a wind turbine, the thesis took a different direction and focused more on utilizing MDCS in drones. When researching similar work, there did not seem to be any work related to using Multiway Dataflow Constraint Systems to program applications other than GUIs. The implementation of MDCS in Rust could emerge new and exciting possibilities [29]. This library is interesting for this thesis, as Rust is a low-level language and could prove very useful for further explorations for usage in robotics and embedded systems. It is common for embedded systems and robotic systems, such as autopilots, to be written in low-level languages, and it seems that the Rust implementation could potentially open up some more doors in this domain. Autopilots must provide quick computations so the vehicle can perform its mission correctly, and the Rust implementation could prove a good fit. Low-level programming languages typically yield faster performance than higher-level programming languages.

The JavaScript version of *HotDrink* could also be suitable but may be more applicable in computations that are not time-critical and that can afford to spend some more time on computation. However, this could relate very well to this project, for further work, through designing graphical mission planners. Mission planners can have a lot of data they should present to the user, and what

the user sees on the screen is highly dependent on the drone sensors and what the user wants to choose in regard to available missions, geo-fencing, planning routes, etc.

Chapter 7

Conclusion

For the system created, which is not too complex and has limited features, the system implemented with MDCS results in overly complex logic and code. This might be due to the chosen architecture, of course, but for a system of this size, a traditional approach would be better and would lead to code that is easier to understand. Applying MDCS to this system adds overhead without any particular benefits. The system implemented in this project worked well with separation of concerns, but not in any way that the traditional approach would lack. With this said, I believe it does not take much more complexity before this negative overhead with MDCS-powered system can diminish and maybe turn more beneficial. This raises an important question: where is the threshold for when MDCS makes sense to add to such a system? Even though no precise answer can be given to this yet, I think we can at least set a threshold at something higher than the complexity and size of this project.

Another consideration is the speed of the system. Although that is out of the scope of this paper, utilizing this approach may result in entirely different runtimes and overhead in terms of application efficiency. Especially for embedded systems, as they typically utilize lower-level languages such as C/C++, using JavaScript and HotDrink might lead to an unforgiving performance loss. However, there is a rust version of HotDrink out there as well, and in the future, more implementations of HotDrink could be released. Maybe some of these versions could be used in such embedded systems, and in the future, perhaps a drone autopilot could be implemented with *hotdrink-rs* [16][29].

There is also more to discover and test with the *HotDrink*-library. Different ways of structuring the constraints, adding a direct reference to rigid body objects, different masks for the constraint inputs, marking constraints optional, etc.

The multiway aspect of Multiway Dataflow Constraint Systems was not utilized in the physics context, as the demos did not reach a level where it was strictly needed. The demos could have been implemented using React's states as well, but that would lead to more "hidden" relations, and the MDCS code is more explicit in this manner.

Using MDCS in such a system is viable, but it might add unnecessary complex-

ity for smaller systems like this. When building this autopilot, adding MDCS through *HotDrink* was most likely disadvantageous regarding code architecture and understanding. Time-critical features of the autopilot, such as hovering, might be better off sticking with the traditional approach unless there is enough computational force and benefits to using MDCS instead. For larger systems, however, It could be worthwhile to explore if they yield better results in terms of readability, assessment of code, and performance when incorporating MDCS.

Chapter 8

Further Work

Much more exploration is needed in the area; this thesis barely scratched the tip of the iceberg. There should be more complex and dynamic demos to explore further how MDCS works in a physics context, and there are many more alterations to be done with the different environments. It would be interesting to see how larger code bases could benefit from this or even a physical quadcopter.

Implementation of MDCS in Rust could be tested and run directly on physical drone hardware. Although creating a full-fledged autopilot is a huge task, it would be exciting to see an autopilot created using an implementation of MDCS.

There is also much more potential to get from using the *HotDrink*-library more extensively. Some of the things in HotDrink that were not tested in this thesis are priority functionality, optional functionality, activations, and promise masks, to name a few. These options could all add up to the usability of using MDCS in an application.

At this time, there is only one constraint in each constraint system. It would be nice to grow the complexity and reach such that multiple constraints could be used. It would also be thrilling to add more focus on how the physical environment (not only objects) in the scene behaves. Wind, rain, and change in gravity would be interesting to work on and see how the constraint systems can be laid up to consider this.

Demos with lower-level details could prove useful in making the demo more realistic. For instance, one could create a drone with four propellers, and instead of applying force to the drone as a whole unit, use it for each propeller. This should also unlock the rotation lock from this demo so that we can observe how the system stabilizes the drone in terms of stabilization, not just height. As well as going into more depth for the vehicles, the focus should also be on widening the types of vehicles added and tested. Quad-copters are not the only vehicles that are much used; we have other robotic systems such as cars and boats as well.

When further working with MDCS in robotic systems, it might reach a point where abstraction of the vehicle itself could be needed. An example would be to express values within the drone sensors, such as battery percentage, as a global

HotDrink-variable that is easy to use. This would make it much simpler for the developer to add these values to a constraint without worrying about updating the values constantly. The variables would be updated between the drone and the abstraction of the drone (*HotDrink* automatically in the background. This could also lead to a new domain-specific language perhaps, where we could utilize these global variables from the drone easily and intuitively.

8.1 Mission Planner

Another use case, which fits well within our domain, is the design of mission planners. They communicate with a drone or other robotic vehicles and enable the user to see live sensor data from the robot. This application also allows for the design of missions, geofencing, route planning, etc. For such applications, the user interfaces can be quite complex. Sensor data that updates, current mission mode, failsafes - many things can contribute to the user interface changing, and MDCS could be well suited to model these. In figure 8.1, one can see a screenshot from a mission planner.

Figure 8.1: A screenshot of a mission planner [2].

We can observe the many elements that make up the page in figure 8.1: live sensor data, the drone's position on a map, information about where it is headed, etc. For implementing GUIs with complex relations that is the task, such as this, MDCS could be an exciting option, and it seems to be an excellent candidate for these types of user interfaces. For Graphical User Interface (GUI)s, such as

this, *HotDrink* is a recommended option [14].

Overall, there is much more to explore in many different directions. Going into finer details while expanding the portfolio of vehicles can enable us to understand better the usability of MDCS in robotic systems. MDCS could aid us in creating safer, more predictable and easier-to-understand systems, that will benefit the robots we create, which again will benefit us humans. In the future, maybe MDCS will have a larger impact on both GUI-programming and systems programming. Further exploration is needed to understand how MDCS can affect robotic systems so that we humans, as users of the systems, can reap the benefits of better and safer systems.

Appendix A

Source code

The source code for this project, along with a demo, is published at: `https://bldl.ii.uib.no/master/2023/lars_oddne_ramstad_juvik/`

The source code for *HotDrink* can be found at the following URL: `https://git.app.uib.no/Jaakko.Jarvi/hd4` I got permission from the author to use *HotDrink*.

The source code for *React* can be found at the following URL: `https://github.com/facebook/react`

The source code for *react-three-fiber* can be found at the following URL: `https://github.com/pmndrs/react-three-fiber`

The source code for *react-three-rapier* can be found at the following URL: `https://github.com/pmndrs/react-three-rapier`

The source code for *drei* can be found at the following URL: `https://github.com/pmndrs/drei`

Bibliography

[1] https://ghloc.vercel.app/PX4/PX4-Autopilot?branch=main. Accessed 11.05.2023.

[2] ArduPilot (https://github.com/ArduPilot). *mission_planner_flight_data.jpg*. https://github.com/ArduPilot/ardupilot_wiki/blob/551ab507445768f4a02ccf2c5d829cb7ce815e89/images/mission_planner_flight_data.jpg. Accessed 01.06.2023. Committed by Hamish Willee (https://github.com/hamishwillee). At the time of image retrieval, licence is CC3.0 BY SA (Attribution-ShareAlike 3.0 Unported), held here https://github.com/ArduPilot/ardupilot_wiki/blob/551ab507445768f4a02ccf2c5d829cb7ce815e89/LICENSE.

[3] Pexels (https://pixabay.com/users/pexels-2286921/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=1866742). *Drone Flying Camera Remote*. https://pixabay.com/photos/drone-flying-camera-remote-control-1866742/. Accessed 23.05.2023.

[4] Roman Ivanyshyn (https://pixabay.com/users/ron2025-16155632/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=5073580). *Robot Vacuum Cleaner Carpet*. https://pixabay.com/photos/robot-vacuum-cleaner-carpet-cleaning-5073580/. Accessed 23.05.2023.

[5] Reuder et al. *SUMO and SAMURAI – GFI/UiB drones for wind energy reserach*. https://www.gceocean.no/media/3914/211110-offshore-wind-conference_sumo-and-samurai_uib.pdf. Accessed 01.06.2023.

[6] *Built-in React Hooks*. https://react.dev/reference/react. Accessed 30.05.2023.

[7] *Compare Roborock Robot Vacuums*. https://global.roborock.com/pages/robot-vacuum-cleaner-compare. Accessed 11.05.2023.

[8] *Constraint Systems*. https://git.app.uib.no/Jaakko.Jarvi/hd4/-/blob/4da3d3b9b27565b9c80d5faa4872e4b24b99eacc/org/constraint-system.org. Accessed 24.05.2023.

[9] *Consumer Drone Market Size, Share & Trends Analysis Report by By Product (Multi-Rotor, Nano, Others), By Application (Prosumer, Toy/Hobbyist, Photogrammetry), By Region, And Segment Forecasts, 2023 - 2030*. https://www.grandviewresearch.com/industry-analysis/consumer-drone-market. Accessed 11.05.2023.

[10] *Determinism*. https://rapier.rs/docs/user_guides/javascript/determinism/. Accessed 09.05.2023.

[11] *Domain Specific Languages*. https://martinfowler.com/books/dsl.html. Accessed 22.05.2023.
[12] *Drone Hits Army Helicopter Flying Over Staten Island*. https://www.cbsnews.com/newyork/news/drone-hits-army-helicopter/. Accessed 16.05.2023.
[13] Charlotte Bay Hasager et al. "Wind Farm Wake: The 2016 Horns Rev Photo Case." In: *Energies* 10.3 (2017). Photo used is cropped out from the figure from the article. Photo is taken by Bel Air Aviation Denmark - Helicopter Services. ISSN: 1996-1073. DOI: 10.3390/en10030317. URL: https://www.mdpi.com/1996-1073/10/3/317.
[14] Magne Haveraaen and Jaakko Järvi. "Semantics of multiway dataflow constraint systems." In: *Journal of Logical and Algebraic Methods in Programming* 121 (2021), p. 100634. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2020.100634. URL: https://www.sciencedirect.com/science/article/pii/S235222082030119X.
[15] *HotDrink*. https://git.app.uib.no/Jaakko.Jarvi/hd4. Permission from author to use.
[16] *hotdrink-rs*. https://github.com/HotDrink/hotdrink-rs. Accessed 16.05.2023.
[17] *HVL Master Thesis Software Engineering Template*. https://www.overleaf.com/latex/templates/hvl-master-thesis-software-engineering-template/qjtzwnzvhrys. Accessed 14.03.2023.
[18] *Introduction to HotDrink*. https://git.app.uib.no/Jaakko.Jarvi/hd4/-/blob/8f47f12dbe576573d0432a461daf47971cc33583/docs/tutorial/tutorial.org. Accessed 24.05.2023.
[19] *Introduction to HotDrink*. http://hotdrink.github.io/hotdrink/howto/intro.html. Accessed 16.05.2023.
[20] *Plane damaged after being hit by York police drone at Buttonville Airport*. https://toronto.ctvnews.ca/plane-damaged-after-being-hit-by-york-police-drone-at-buttonville-airport-1.5554617. Accessed 16.05.2023.
[21] Poimandres. *drei*. https://github.com/pmndrs/drei.
[22] Poimandres. *leva*. https://github.com/pmndrs/leva.
[23] Poimandres. *react-three-fiber*. https://github.com/pmndrs/react-three-fiber.
[24] *rapier*. https://rapier.rs. Accessed 19.05.2023.
[25] *React*. https://reactjs.org.
[26] *React*. https://react.dev. Accessed 27.05.2023.
[27] *react-three-rapier*. https://github.com/pmndrs/react-three-rapier. Accessed 19.05.2023.
[28] *Separation of Concerns*. https://help.sap.com/doc/abapdocu_753_index_htm/7.53/en-US/abenseperation_concerns_guidl.htm. Accessed 19.02.2023.
[29] Rudi Blaha Svartveit. "Multithreaded Multiway Constraint Systems with Rust and WebAssembly." In: (2021), p. 123. URL: https://bora.uib.no/bora-xmlui/handle/11250/2770614.
[30] *three.js*. https://threejs.org.
[31] *What is a digital twin?* https://www.ibm.com/topics/what-is-a-digital-twin. Accessed 08.05.2023.