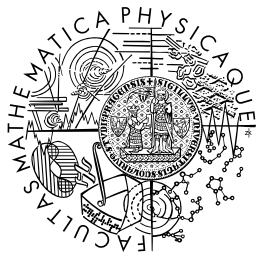

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Sára Goldscheiderová

Interactive pandemic simulation to encourage critical thinking

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Tomáš Petříček, Ph.D.
Study programme: Computer Science

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague date 2024–05–08 Sára Goldscheiderová

I would like to thank my supervisor, Mgr. Tomáš Petříček, Ph.D., for his time and guidance throughout the process of writing this thesis. I would also like to thank my family, friends, boyfriend, and cats for their emotional support and encouragement.

Title: Interactive pandemic simulation to encourage critical thinking

Author: Sára Goldscheiderová

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Tomáš Petříček, Ph.D., Department of Distributed and Dependable Systems

Abstract: The complexity of pandemic simulations often makes them opaque and difficult to understand for the general public. Critical thinking is essential for understanding the results of these simulations, but the current methods are lacking in this regard. They commonly leave people skeptical and unable to comprehend the implications of the simulations. This thesis aims to design an interactive pandemic simulation that encourages critical thinking and implement a prototype of it. Through this simulation design, we can illustrate how future models can be made more socially beneficial and how they can be used to educate the public.

Keywords: simulation data visualization

Název práce: Interaktivní simulace pandemie pro podporu kritického uvažování

Autor: Sára Goldscheiderová

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Tomáš Petříček, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Složitost simulací pandemií způsobuje, že jsou pro veřejnost často nejasné a obtížné k pochopení. Přestože je kritické uvažování nezbytné pro správné pochopení výsledků těchto simulací, současné metody v tomto ohledu selhávají. Důsledkem toho lidé simulacím často nedůvěřují, nechápou je, a neví, co si z nich ve skutečnosti odnést. Tato práce si klade za cíl navrhnout interaktivní simulaci pandemie, která podporuje kritické uvažování, a implementovat její prototyp. Skrz ní pak můžeme ukázat, jak by se mohly budoucí modely simulací stát společensky přínosnějšími.

Klíčová slova: simulace vizualizace dat

Contents

Introduction

Computer simulations are programs that use mathematical models to mimic processes in the real world. We can differentiate three main purposes for which they are typically used: understanding, predicting, and explaining [1].

To illustrate the difference between the three purposes, consider the following example. Imagine we were to simulate the atmosphere above a Midwestern plain. The simulation could be used to understand the structure of a severe thunderstorm, forecast weather, or serve as a tool to explain why storms sometimes split in two [1].

One of the areas that heavily relies on computer simulations is computational epidemiology. In simple terms, computational epidemiology develops and uses simulation models of synthetic social contact networks to understand and control the spread of disease in populations [2]. The results of these simulations are then analyzed when making decisions about public policies. Epidemiological models focus on accurate use of data and complex mathematical algorithms to obtain the most realistic results possible.

However, it is also possible to imagine a different approach to simulations, one where we focus more on the user experience and less on the accuracy of the data. One of the possibilities is to create a simulation that encourages critical thinking. In *Explorable Explanations* [3], Victor mentions a few ideas on how to change people's relationship with text. He introduces the concepts of reactive documents, explorable examples, and contextual information. In some aspects, these designs can be applied to simulations as well.

Some examples of simulations that encourage critical thinking can be found in Washington Post's article [4] *Why outbreaks like coronavirus spread exponentially, and how to "flatten the curve"*, or R2D3's article [5] *Making sense of COVID19 through simulations*. We will discuss these examples and how they're designed in more detail in the following chapters.

Contributions

The goal of this thesis is to create an interactive pandemic simulation that encourages critical thinking rather than an attempt to create an accurate model of the spread of COVID-19. In order to achieve that, we analyze various design goals and visualisation options, compare them against existing simulations, and then design our own model. The environment of our simulation strives to inspire our reader to be an active one. An active reader is someone who isn't just passively sponging up information, but asks questions, considers alternatives, and questions the author and their assumptions [3].

The contributions of this thesis are as follows:

1. We analyze various design goals and visualization options for pandemic simulations.

2. We design an interactive pandemic simulation that encourages critical thinking.

3. We implement a prototype of the simulation.

Chapters 1 and 2 cover the analysis of simulations and the design goals we want to set for our own simulation. Chapter 3 describes the design of our simulation from a high-level perspective as well as the details of our implementation. Final thoughts and possible future improvements are discussed in Chapter 4.

Chapter 1

Background

This chapter provides a theoretical foundation for the rest of the thesis. We define simulations and visualizations, explore the impact they have on education and critical thinking, and discuss their trustworthiness. This is followed by an overview of available interactive simulation tools and libraries used in this thesis. In the rest of the thesis, we draw from existing simulations and visualizations reviewed in this chapter to create our own interactive simulation that encourages critical thinking.

1.1 Simulations

We have already briefly touched upon the topic of computer simulations in the introduction, where we established that computer simulations are programs that use models to mimic real life. However, if we look at different sources, we discover that definitions can vary. Let us look at the definition from a different perspective to ensure that we have a clear understanding of what a computer simulation is. In order to do that, we need to take a step back and consider simulations in general.

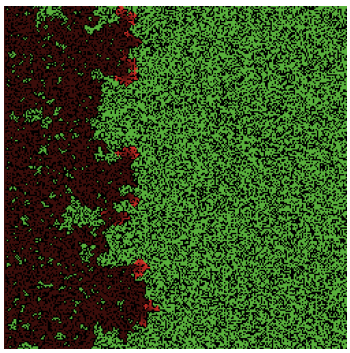

Figure 1.1 A simple simulation of fire spreading in NetLogo [6].

A simulation is a system with dynamical behavior that mimics another system. It does so in a way where we can learn something about the mimicked system while observing the simulation [1]. However, matching the simulation with real-world reality is not always possible. A good example is mentioned in the book *An introduction to computer simulation* [7] where the authors mention simulations of nuclear weapons. Testing nuclear weapons in real life is not only dangerous, but also damages the environment, hence computer simulations have been developed as a safer alternative. That way new weapons can still be tested without actually detonating them.

Thus if we first define a simulation as mentioned above, an alternative definition of computer simulations is also possible: a simulation carried out by a computer program [1].

1.1.1 Simulations in Education

Once a simulation is placed in a specific context, the definition can be further modified. In the book *Games and simulations in online learning: research and development frameworks* [8] one of the definitions of a computer simulation introduced by the authors includes the following: "..., which the student has to act upon." Since we are trying to encourage critical thinking, having the user interact with the simulation and act upon what they see is crucial. Let us explore simulations in education further to gauge their relevance to what we're trying to achieve.

As we have already mentioned, simulations can be used for explanatory purposes, through which they can become powerful tools that help students understand complex systems. Many of the "explorable explanations" [3] created so far are actually of a pedagogical nature, although that wasn't Victor's intention.

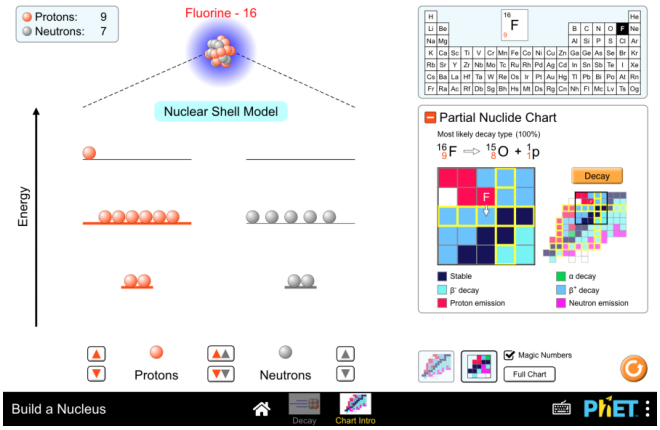

Figure 1.2 An educational simulation of a nucleus by PhET.

One of the applications of simulations in education are simulation games. In "Simulation in education and training" [9] the authors set certain criteria for a simulation game to be considered educational and help students learn. The game must be interactive, have theoretical grounding and have some random elements so that it can be replayed multiple times with different experiences. They have also identified specific examples of simulation games, such as "SimCity" and "Microsoft Flight Simulation 98."

Multiple authors also studied the effects of simulation on learning. The responses are varied. Research has shown that using simulations as a supplementary teaching tool impacts critical thinking and reasoning in a positive way. The way students interact with simulations leads to a more active reception of knowledge [10] and is also associated with improvements in motivation and class attendance [9]. However, some professionals disagree. In the book *Simulation and its discontents* [11] we're faced with opinions such as "simulations allow students to get answers without understanding the underlying principles." This suggests that students aren't actively thinking about the simulation, but rather blindly trusting it.

1.1.2 Trustworthiness of Simulations

Computer simulations are rapidly gaining popularity in various fields. With their growing importance and usage, the concerns about their trustworthiness grow as well, especially when it comes to simulations that are used to further our knowledge of something. The core of the issue is whether the results are accurate enough to be trusted [1]. This so called "can you trust it?" problem is most prevalent in socio-technical systems. These systems are defined by having both physical and human parts, e.g., a city, or a road system [12].

It is clear that students of architecture and planning will be one of the people using simulations based on socio-technical systems. Let us circle back to *Simulation and its discontents* [11] where they summarize the opinions of professors in these fields. They argue that students have no other choice than to trust the simulations, consequently trusting the programmers who wrote them. This is not ideal, as it does not encourage critical thinking at all.

An example of the lack of critical thinking is included in the book as well. Albeit not a student of architecture or planning, a thirteen-year-old player of "SimCity," one of the simulation games we mentioned earlier, thought that since raising taxes in the game led to riots, it must be the same in real life. If the player had thought critically about the simulation game, they would have questioned the rule and perhaps even compared it against historical data of real cities.

As we can see, we definitely can't fully trust everything we see in simulations to be applicable in real life. However, that doesn't mean we shouldn't trust

simulations at all. “If you do not trust a carefully executed simulation, you probably have less reason to trust anything else, including the way you currently make decisions.” [12] Simulations that ensure the users are actively thinking about the results and questioning them are the middle ground that we believe we should strive for.

1.2 Visualizations

Visualizations can be defined as mappings between data and a visual representation. Similarly to simulations, alternative definitions can be found in different media, as it is more of an umbrella term for several areas such as information visualization, scientific visualization, information design, etc., rather than a specific field [13].

While visualizations often go hand in hand with simulations, they aren’t restricted to each other. Visualizations can be used to represent all kinds of data. So while we can use them for simulation results, we can also use them for data we’ve collected from real life and simply want to present in a more digestible way.

An example of this is shown in figure 1.3, where we can see a visualization of deaths due to COVID-19 over time. The visualization is in the form of a simple line graph. It is easy to understand for the general public, but doesn’t provide any room for critical thinking.

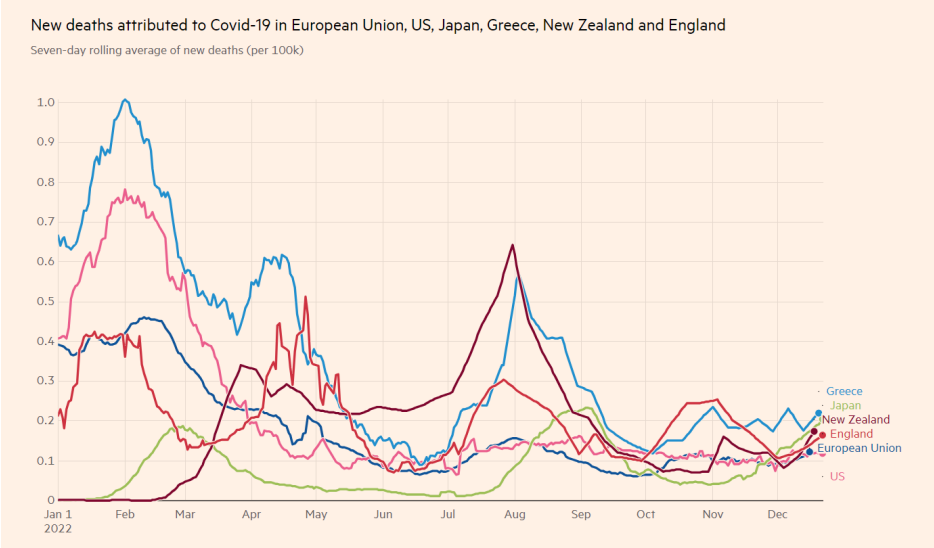

Figure 1.3 A visualization of deaths due to COVID-19 [14].

1.2.1 Visualizations and Critical Thinking

According to Ge, Cui, and Kay [15], badly designed visualizations, whether on accident or on purpose, can easily lead to misinformation. They even mention examples, one of which is shown in figure 1.4. It depicts a visualization of COVID-19 cases by the Departmen of Public Health of Georgia.

Instinctively, we would assume that the visualization is sorted chronologically. However, it is sorted by the number of cases in descending order, leading to a false impression of the situation at the time. This is one of the many reasons why it is important, necessary even, to actively think about the visualizations we see and question them.

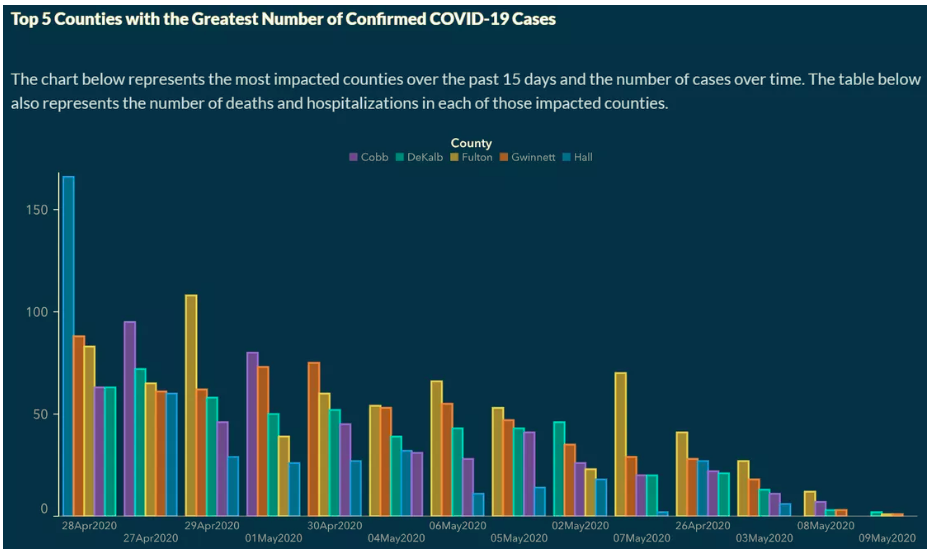

Figure 1.4 A visualization misinformation of COVID-19 cases by Georgia DPH.

There's only so much we can do to prevent others from creating misleading visualizations. But if more authors prompted their readers to think critically about their visualizations, we could potentially encourage them to question other visualizations as well. We've already mentioned some concepts from *Explorable Explanations* [3] in the introduction. Now let's look at some concrete examples.

One of the mentioned concepts was a reactive document. Victor's own creation *Ten Brighter Ideas? An Explorable Explanation* [16] serves as a good example of this. It allows the reader to change certain parameters and see how their changes affect the results in real time. Another example, this time of an entirely different concept that we haven't mentioned yet, can be found in the article *Accounting for democracy* [17]. There, the authors let the reader guess the graphs before revealing them. This forces the reader to actively think about the data.

A visualization alone can encourage critical thinking, but for our purposes, that isn't enough. The spreading of a virus is more complex, doesn't behave the same way every time, and can't be represented statically. We need to combine it with a simulation to fully showcase its behavior.

1.2.2 Visualizations with Simulations

Visualizations of simulations need to map the model, the simulation itself, and its output onto a visual representation in context of the real world system being simulated. The success of the simulation often depends on how the visualization is designed [18]. A good visualization can help the user see important patterns and trends, which leads to a better understanding of the simulated system [19].

Poorly designed visualizations can be the result of focusing too much on the simulation and leaving the visualization as an afterthought, not adhering to the principles of good visualization design, etc. This can lead to misunderstandings and misinterpretations of the simulation results [18].

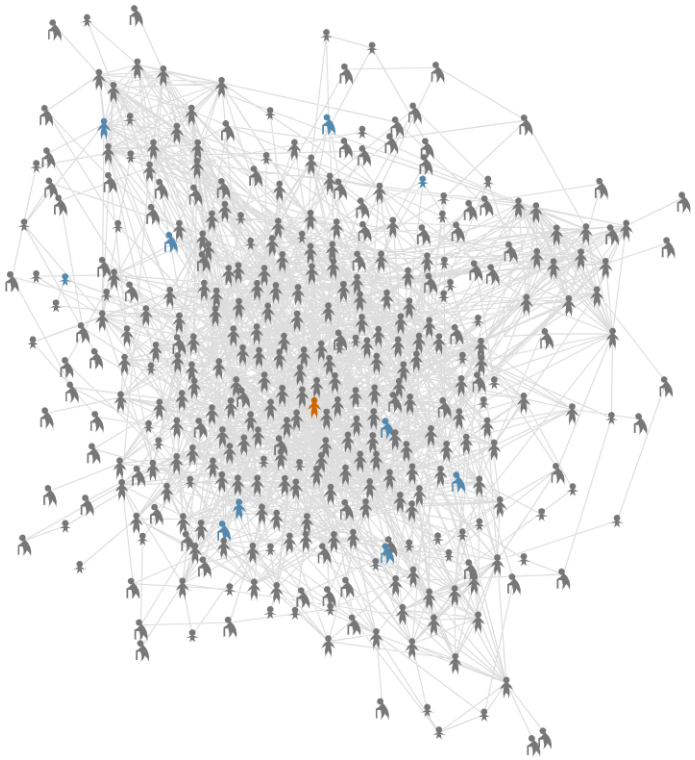

Figure 1.5 An example of a social network from a COVID-19 simulation [5].

Once we start the simulation from figure 1.5, the nodes change colors depending on their state. We're supposed to see the virus spread through the network,

see how many people need to be hospitalized, how many recovered, died, etc. But due to the complexity of the network, it's difficult to distinguish their connections and the simulation becomes hard to follow. It might have been more useful to simplify the network. This is why we need to think about the design of the visualization in the context of what we're trying to highlight to our audience.

1.3 Interactive Simulations

Interactive simulations are simulations that allow the user to interact with them in some way. This can be done by changing parameters, moving objects, etc. Ideally, such interactions receive immediate feedback, allowing the user to see the results of their actions in real time.

The goal of this thesis is to create an interactive simulation that encourages critical thinking. After researching simulations and visualizations separately, we have a better understanding of how they work best, what possible problems we might face, and how we can use them to prompt critical thinking. Now we need to look at how we can combine them efficiently.

For simulating the spread of a virus, agent-based simulations are often used. Agent-based simulations have agents as the main entities. Each agent has its own set of local rules that it follows. The agents then interact with each other and their environment based on these rules [1]. Among some of the options are NetLogo [20], AgentScript [21], and MASON [22].

1.3.1 NetLogo

In "Netlogo: A simple environment for modeling complexity" [23] the authors introduce NetLogo, a multi-agent programming language and modeling environment. It is simple enough for non-professional students and researchers to use and is designed with both teaching and research in mind.

There are several agents in NetLogo - turtles, patches, and the observer. Turtles move around and interact with each other. Different variables and behaviors can be assigned to them, creating unique "breeds" of turtles, as the authors call them. Patches are the environment in which the turtles move in. We can also change them to our liking. Only one observer exists in the simulation and usually issues commands to the other agents.

NetLogo has an extensive open source library of models from which we can draw inspiration. One of the examples is a virus spreading simulation, as seen in figure 1.6. We have considered using NetLogo for our simulation, but it doesn't allow for the level of interactivity we're aiming for. In addition, it's not very user-friendly for the general public.

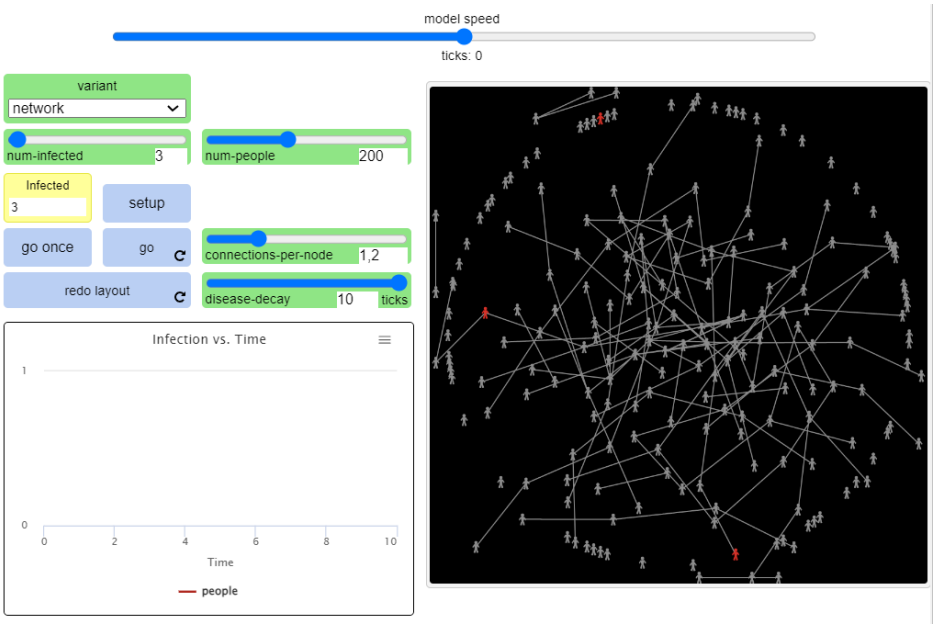

Figure 1.6 An interactive simulation of virus spreading in NetLogo.

1.3.2 AgentScript

According to their website [21], AgentScript is an open source JavaScript library for creating agent-based simulations inspired by NetLogo. It is designed to be minimalistic and uses the MVC pattern.

Similarly to NetLogo, AgentScript also has three types of agents. Two of them are the same - turtles and patches. Turtles are the agents that move around and patches are the environment. The third type isn't an observer, but links. Links are connections between turtles. This can be useful for example for networks.

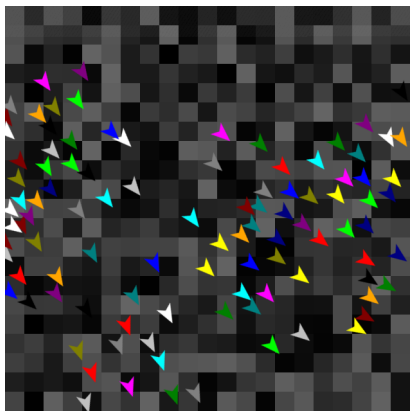

Figure 1.7 An example of a flocking simulation in AgentScript.

From the examples provided on their website, it appears that AgentScript is focused on the movement of the agents rather than the connections between them. That is not what we're looking for, so we have decided not to use AgentScript.

1.3.3 MASON

MASON is introduced in "Mason: A multiagent simulation environment" [22] as an open source simulation toolkit and visualization library. The main idea behind MASON are swarm multi-agent simulations. It is designed mainly for professional researchers and developers.

Its computational entities are called agents. If they are in the environment, they are referred to as embodied agents. However, MASON doesn't require agents to be in the environment. Fields are optional and associate objects or values with locations in the environment.

MASON doesn't meet the criteria we're looking for either. It is a minimal model library and in order to create the social network simulation we're aiming for, we would need to use an existing extension or make it ourselves. As mentioned, MASON is designed mainly for research. It isn't user-friendly for the general public and lacks the desired interactivity.

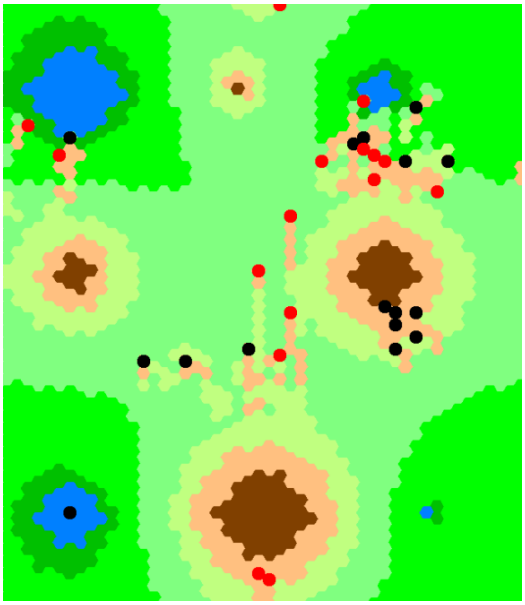

Figure 1.8 An example of a wetlands visualization in MASON.

1.4 Libraries

To best showcase the spreading of a virus, we need a visualization of a social network. A graph where the nodes represent people and the edges represent connections between them is the most suitable for this purpose. Ideally, we would like to have the option to easily deploy the simulation on the web. We have chosen *D3.js* [24] as the library for this task.

1.4.1 D3

Bostock, Ogievetsky, and Heer [25] define D3 (short for Data-Driven Documents) as a non-traditional visualization library. It is an open source software focused mainly on the generation and efficient manipulation of web documents with data. The basic functionalities of D3 include selecting elements, binding data to said elements, modifying elements based on the data, and transitioning elements in response to user input.

The library itself is quite complex. It is more of an ecosystem of modules that can be used together than a monolithic library. This makes it more difficult to create simple visualizations, as it requires more code than people might expect. On the other hand, it allows for more flexibility and customization [24].

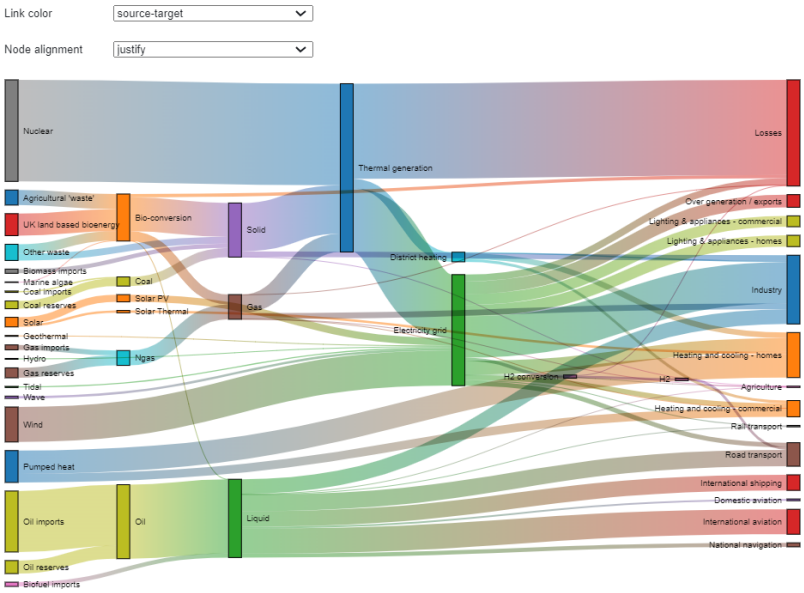

Figure 1.9 An example of a D3 visualization.

D3 also has an extensive library of examples. One of them is shown in figure 1.9. It is a good place for beginners to start and explore the possibilities of D3. That

is what we did as well. We have taken the code from one of the examples and played around with it to learn how certain parts of D3 work. The conclusion is that D3 is the best choice for our purposes.

1.4.2 Alternatives

While searching for a suitable library, we came across several possible alternatives. Two of them were taken from *Interactive data visualization for the web: an introduction to designing with D3* [26] - *Arbor.js* [27] and *Sigma.js* [28]. The last one is *El Grapho* [29].

Arbor.js uses web workers and jQuery. The framework is much simpler than D3, but it didn't seem to have any outstanding features that would convince us to use it. It lacked the flexibility and customization that D3 offers. After studying the documentation, we have decided that D3 is the better choice.

Sigma.js uses WebGL. It has some attractive features, such as letting the users create and manipulate graphs or letting them explore the neighbouring nodes by hovering over a node. However, on their website [28], they mention that the library is more fit for larger graphs and that custom rendering is more difficult. We will be working with smaller networks and more customization, so we have decided not to use Sigma.js.

El Grapho also uses WebGL. From the examples they provided, it seems that the library has a lot of potential, but faces the same issues as Sigma.js, meaning it is more fit for larger graphs and doesn't offer as much customization as D3. Thus we have decided not to use El Grapho either.

Chapter 2

Requirements Analysis

In this chapter, we will analyze and define the requirements for our simulation. We will start by defining the main design goals and then we will discuss how to apply them to our simulation. In the last part of this chapter, we will analyze how existing simulations meet these requirements.

2.1 Design Goals

From what we've learned about interactive simulations in Chapter 1, we concluded that there are three main design goals that we want to set for our simulation.

Empowerment The user should be empowered to change the simulation. This will allow them to test their assumptions as well as the assumptions of the author of the simulation.

Explanation The simulation should provide insight into how the virus spreads. This will help the user understand the dangers of pandemics and the consequences of their actions.

Understanding The simulation should be easy to understand and use. This will allow the user to focus on the main questions and not get lost in the details of the simulation.

Combining these three design goals will help us create a simulation that fulfills our main purpose. However, we have to be careful with how we implement them. We need to find a balance between providing the user with enough information to understand and benefit from the simulation, but not overwhelming them with too much information and too many options.

2.2 Implementation Principles

Burton and Obel [30] talk about how important it is to understand the questions our simulation should answer before we start designing it. The simulation model should not be overly complex, instead, it should remain as simple as possible while still addressing the main questions.

In order to do that, we need to identify some implementation principles for our simulation. These principles will help us understand what we need to focus on and what we can disregard later on. Since we are building a simulation that encourages critical thinking, we need to make sure that we test the user's assumptions. We concluded that the following three implementation principles are the most important for our purposes:

1. Visual Representation The simulation should be visually appealing and easy to understand. The user should know what is happening in the simulation at all times. If the user doesn't understand the simulation, they won't be able to think about it critically.

2. Interactivity The user should be able to interact with the simulation. They should be able to test out different scenarios. This should encourage them to think about the different possibilities and outcomes.

3. Immediate Feedback The user should see the results of their actions immediately. This should help them understand the consequences of their decisions. Additionally, it should prompt them to question whether certain actions truly have the depicted effects.

We've decided to omit accuracy from our implementation principles. The intention behind this simulation is not to present a realistic model of what happened during the pandemic, but rather let the user explore different situations and gain a better understanding of how viruses spread.

2.2.1 Applying the Implementation Principles

The principles we've identified so far are quite high-level. They don't provide us with a clear idea of how to implement them in our simulation. This subsection will focus on analyzing possible implementation strategies for each of the principles.

A crucial part of 1. Visual Representation is the visualization of people and their connections. The spreading of a virus is a complex process. Choosing to make an equally complex visualization that shows as many details as possible might result in the user getting overwhelmed. We need to keep the visualization

simple and the simulation slow enough for the user to see the spread of the virus clearly. The main information we need to convey is the health status of each person and the connections between them. We can achieve this by choosing the right shapes and colors.

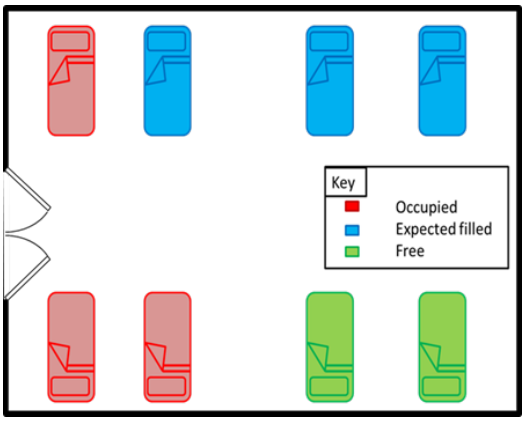

Figure 2.1 A visualization that might be hard to read for color-blind users [31].

We have decided to represent people as circular nodes and their connections as lines (edges) between them. The color of the node represents the health status of the person. We use two colors: gray for healthy people and red for infected people. It may appear more natural to use green for healthy people, but it could cause similar problems as in figure 2.1. Red-green color blindness is the most common form of color blindness and affects around 8% of the population [31].

Figure 2.2 Our representation of healthy and infected people.

The connections between people are also colored. Green connections are family members, purple are friends, blue are coworkers/classmates, and gray are strangers. This way the user can see how the virus spreads through different types of relationships. Additionally, we thicken the lines of infected individuals to make the spread of the virus easier to follow. In figure 2.3, you can see that the gray connection (stranger) is further away from the infected person than the rest. By pushing weaker connections to the background, we can make the graph easier to read.

Figure 2.3 All types of connections.

To fullfill the 2. Interactivity principle, we need to provide the user with a set of tools that allow them to change the simulation. The two main points of interaction are the ability to modify the social network and the ability to change how the virus spreads.

There are many different options for how the user can change the simulation. We have decided to let the user modify the amount of nodes in the graph and the initial number of infected people. This allows them to understand the spreading of the virus in communities of different sizes.

The user can also choose the aggressiveness of the virus and modify the spreading probabilities through different types of regulations. Each regulation has a different effect on the spread. The user can see a brief summary of the regulation's effect when they hover over the text. This allows them to see how different actions can affect the spread of the virus. It also gives them the space to think critically about whether the author's assumptions about the effects of the regulations are correct.

We give the user the option to choose from a set of predefined scenarios or create their own. This challenges the user to think about how the virus spreads under various conditions.

All of these interactions are done through a simple user interface that meets the 1. Visual Representation principle. We use sliders, radio buttons, checkboxes, and dropdown menus to make the interaction as intuitive as possible.

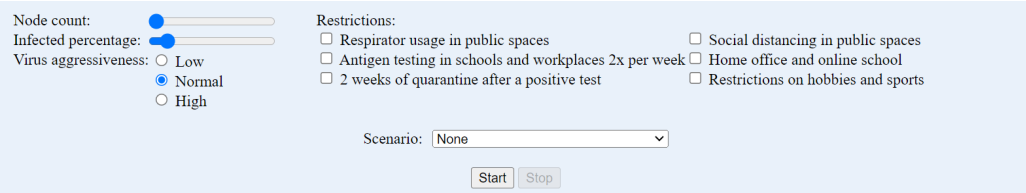

Figure 2.4 The user interface for the simulation.

3. Immediate Feedback is achieved by giving the user a visual confirmation of their actions. When the user changes something in the simulation, they can

check the results immediately. Changes made to the social network should be reflected in the graph. Adjusting how the virus spreads should be visible in the number of infected people. This helps the user understand the consequences of their actions.

We have decided to generate a new graph every time the user changes the social network. Modifying the graph in real-time would also be possible, but it would be more difficult to implement. To help the user see the number of infected people, we display a chart below the simulation graph. This chart shows the number of newly infected people over time. Comparing the chart of two different scenarios makes it easier for the user to analyze the trends and identify what caused them.

2.3 Existing Simulations

We have chosen three existing simulations to analyze and compare against our design goals. Both The Washington Post's simulation [4] and R2D3's simulation [5] are used in articles that attempt to educate the public about the spread of the virus. The third simulation is from the Imperial College team [32]. It was used to make predictions and played a role in the policy-making process in the UK.

The Washington Post's simulation is the simplest of the three. As can be seen in figure 2.5, the agents do not have any connections between them. They spread the virus by randomly moving around the screen. There are no interactive elements in the simulation, forcing the user to rely on the author's assumptions. However, thanks to its simple design, it is easy to understand. It shows different scenarios and explains how certain decisions can affect the spread of the virus. The user can clearly see how the virus spreads through the population.

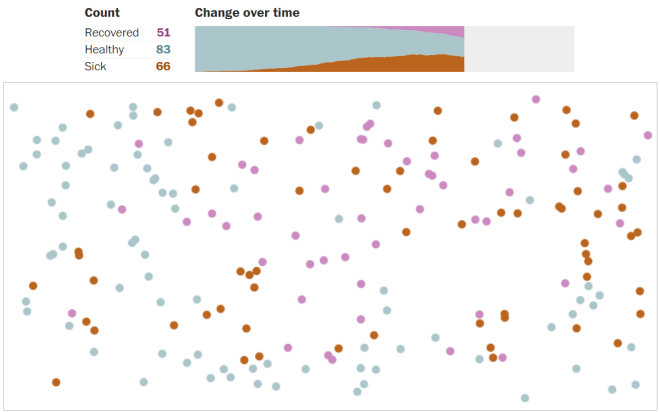

Figure 2.5 The Washington Post's simulation.

R2D3's simulation is more complex. It is the only one of the three that allows the user to interact with the simulation. The user can change some parameters of the simulation, such as which connections should abide by social distancing rules. It also provides the user with explorable examples [3], such as a chart where the user can adjust the start date of social distancing. This allows the user to build an intuition about how different actions can affect the spread of the virus. However, the visualization of the simulation is complex. It is hard to see the connections between people and the simulation is quite fast, making it difficult to follow the spread of the virus.

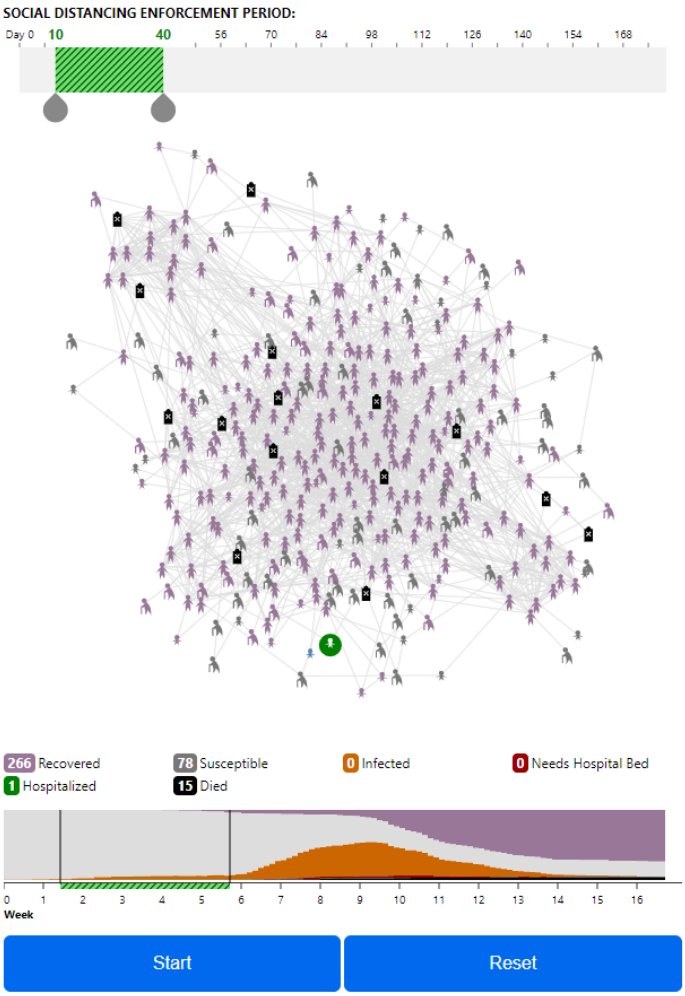

Figure 2.6 R2D3's simulation.

The Imperial College simulation is the most complex of the three, as it strives to predict the spread of the virus accurately. It does not have a visualization of

the spread of the virus and is not interactive. The user can only see the results of the simulation, which offer little insight into how the virus spreads. An example can be seen in figure 2.7.

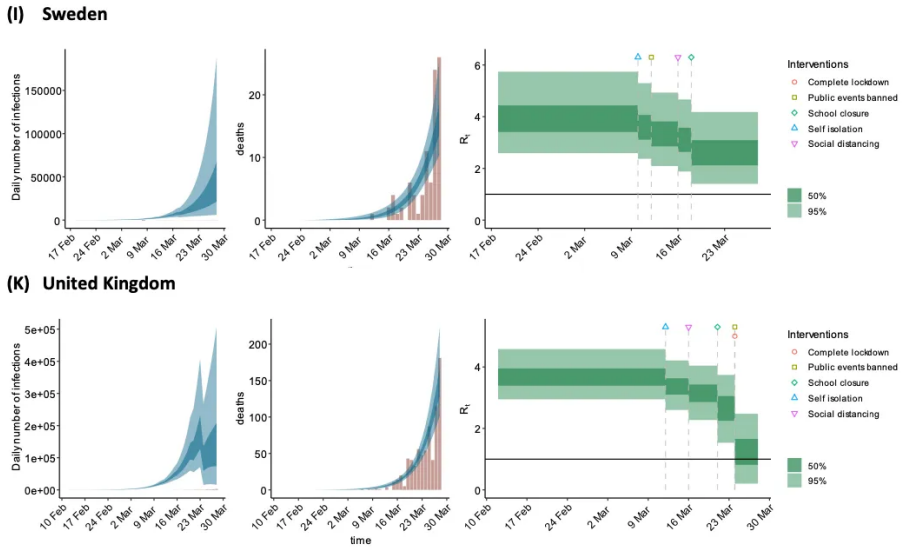

Figure 2.7 Diagrams from the Imperial College. [33]

After summarizing the analysis of the existing simulations in table 2.1, we can see that each simulation fulfills different design goals, but none of them fulfill all three. This supports our claim that current simulations do not provide the user with the right tools to think critically about the spread of the virus during a pandemic.

Simulation	Empowerment	Explanation	Understanding
The Washington Post	No	Yes	Yes
R2D3	Yes	Yes	Partially
Imperial College	No	Partially	No

Table 2.1 Three existing simulations and their fulfillment of the design goals.

Chapter 3

Designing a Simulation

This chapter describes the design of the simulation. It is divided into two main sections: the model and the implementation. In the first part, we define what components are present in the simulation and how their states change, what the configuration options are, and the main functionalities. In the second part, we describe the implementation of the simulation, including the structure of the code, the use of D3 for visualization, and the development tools used.

3.1 Model

Models are used to represent real-world systems in a simplified way. In our case, we are modeling the spread of a virus in a population. That is best done by a discrete event simulation, which is a simulation where the state of the system changes at discrete points in time [34]. Our model is made up of two main components: agents (nodes) and edges, each of which has its own attributes that define its state, as shown in figure 3.1.

The model is purposefully kept simple to make it more accessible for non-programmers, as our goal is to create a tool that may be used by the general public. However, possible extensions are later mentioned in section 4.3. We will discuss what each of the components represents and how their states change in the next subsection.

The state of the **Model** consists of:

- Collection of Nodes
- Collection of Edges

Node consists of:

- Infected (Boolean)

Edge consists of:

- Source Node
- Target Node
- Edge Type

Edge Type can be one of the following:

- Family
- Friend
- Classmate or Coworker
- Stranger

Figure 3.1 The state of the model.

3.1.1 State of the Model

The model is a representation of the population. Nodes represent individuals in the population, while edges represent connections between them. Figure 3.1 shows that the state of the model is defined by the state of the nodes and edges. It changes at each tick, which represents the passage of time.

Nodes The state of a node is represented by a boolean value, which determines whether the node is infected or not. The state of a node can only change from not infected to infected. This change can happen under the condition that at least one of its connected nodes is already infected.

Edges The state of an edge is represented by three attributes: source node, target node, and edge type. The source and target nodes are the nodes connected by the edge and do not change. The type of the edge determines the probability of being infected by the neighbouring node and does not change.

3.1.2 Configuration

The user has the option to modify the configuration of the simulation. The user can set the number of nodes in the graph and configure their initial state by changing the percentage of infected nodes. The user can also adjust the probability of the success of the spreading event. This probability is made up of two components: the spreading probabilities for each edge type and the spreading rate.

Spreading probabilities have a base value depending on how close the relationship between the nodes is. The user can modify these probabilities by choosing different restrictions. The probabilities are recalculated each time the simulation is started.

The spreading rate is the aggressiveness of the virus. The higher the spreading rate, the more likely the virus is to spread. The user has three options to choose from: low, normal, and high. The spreading rate is recalculated each time the simulation is started.

3.1.3 Main Functionalities

The most important functionalities of the simulation are data creation and virus spreading. We will outline the basic steps through pseudocode. A more detailed description of our implementation can be found in the next section.

For the simulation to work, we need to create the necessary data. If the user doesn't provide their own data for the graph creation, a random number of nodes is generated. The default percentage of infected nodes is 10%. The edges are then created by randomly connecting the nodes. The type of the edge is also randomly assigned. If some nodes happen to have no connections after the edge creation, they are removed.

Algorithm 1 Pseudocode for the data creation.

```
if nodeCount is 0 then
    nodeCount ← random number between 10 and 100
end if

for i = 0 to nodeCount do
    state ← state based on percentageOfInfected

    create new Node with state
    nodes ← nodes with new Node
end for

for i = 0 to linksAmount do
    sourceNode ← random node
    targetNode ← random node different from sourceNode
    type ← random type
    value ← value based on the states of nodes

    create new Link with sourceNode, targetNode, type, value
    links ← links with new Link
end for

call deleteUnlinkedNodes
```

The core of the simulation is the spreading of the virus. Once the simulation is started, the virus attempts to spread until all nodes are infected. We iterate over the edges of infected nodes and check if the virus can spread to the neighbouring node. If the conditions are met, we try to infect the node based on the spreading probabilities and the spreading rate.

Algorithm 2 Pseudocode for the spreading of the virus.

```
if all nodes are infected then
    stop the simulation
end if

infectedLinks ← links with an infected node
newlyInfectedNodes ← empty array

for link in infectedLinks do
    if link node is infected and link node not in newlyInfectedNodes then
        call tryInfect with other link node
        if successful then
            newlyInfectedNodes ← newlyInfectedNodes with other link node
        end if
    end if
end for
```

3.2 Implementation

The simulation is a simple web application written in JavaScript that uses the D3 library for its visualizations and number generation. As there is no need for a backend, the simulation can be run locally. There are three main files in the project: `index.html`, `style.css`, and `main.js`. The `index.html` file contains the structure of the page, the `style.css` file contains the styling, and the `main.js` file contains the implementation of the simulation. We will focus mostly on the `main.js` file.

3.2.1 Structure of the Code

The code is divided into several classes, each of which represents a different part of the simulation. They are not strictly mapped to the components of the model, as we don't need a separate class for edges, for example. We tried to keep the names as descriptive as possible to make the code more readable. Edges are represented as links in D3, which is why we chose to use that term throughout the code for consistency.

The class `GraphNode` represents a node in the graph. The constructor takes a boolean value `infected` as an argument, which determines whether the node is infected or not. It has no methods, as the simulation later mutates the nodes. Calling methods on the mutated nodes would result in errors, since they are no longer instances of the `GraphNode` class. That is why the function `tryInfect`

must remain outside the class.

The function `tryInfect` is used to try to change the state of a node. If the node is already infected, we return false, as re-infecting a node is not possible in our model. If the node is not infected, we use the `randomBernoulli(p)` function from D3. This function returns true with probability `p` and false with probability `1 - p`. In our case, the probability is calculated as `p = probability * spreadRate`. If the function returns true, we infect the node and return true. Otherwise, we return false.

Listing 1 Node representation and state changing function.

```
// === NODES ===

// Represents a node in the graph
class GraphNode {
  constructor(infected)
}

// Tries to infect a node with a given probability
// and spread rate
function tryInfect(node, probability, spreadRate)
```

The class `Simulation` represents the simulation itself. The only property of the class is `infectedAmounts`, which is an array that stores the number of newly infected nodes after 5 ticks. This array starts with helper values `{x:0, y:0}`, `{x:5, y:0}`. This ensures that the chart starts at 0 and displays the first value after 5 ticks. The class has a considerable number of methods, as it is responsible for the main functions of the simulation.

Listing 2 Simulation representation.

```
// === SIMULATION ===

// Represents the simulation
class Simulation {
  ...
}
```

We need to create the data for the graph first, which is done by the `createData` method. This method calls the `decideNodeCount`, `createNodes`, and `createLinks` methods.

The `decideNodeCount` method takes the `nodeCount` parameter from the user input. If the parameter is 0, the method generates a random number of nodes

between 10 and 100. This ensures that the graph is neither too small to properly display the spreading of the virus nor too large to confuse the user.

The `createNodes` method is implemented as a simple for loop that creates `nodesAmount` instances of the `GraphNode` class and pushes them into the `nodes` array. It uses the same probability function as `tryInfect`, but this time it is used to determine the initial state of the node. Hence `p = percentageOfInfected`.

The `createLinks` method creates approximately `nodesAmount * 2` edges. The source and target nodes are chosen randomly. We have to ensure that they are not the same, as we don't want self-connections. It could cause unconnected nodes to stay in the graph, as they would not be removed by the `deleteUnlinkedNodes` method. The type is generated randomly.

The `deleteUnlinkedNodes` method is used to remove nodes that have no connections. This is done to make the graph easier to read. We create an array of node indexes. By iterating over the edges, we delete the indexes of connected nodes from the array. We then reverse the array and delete the nodes with the indexes from `nodes`. Reversing the array is necessary, as it would cause index shifting otherwise.

Listing 3 Data creation methods.

```
// === DATA CREATION ===

class Simulation {
  ...

  // Creates the data for the graph
  createData(nodeCount, percentageOfInfected)

  // Decides the number of nodes in the graph
  decideNodeCount(nodeCount)

  // Creates the nodes of the graph
  createNodes(nodes, nodeCount, percentageOfInfected)

  // Creates the edges of the graph
  createLinks(links, nodes, nodeCount)

  // Deletes nodes that have no connections
  deleteUnlinkedNodes(nodes, links)

  ...
}
```

After the deletion of unlinked nodes, we need to reindex the rest. This is done

by two methods: `reIndexNode` and `reIndexLinkNodes`. The first method counts how many `unlinkedIndexes` are lower than `currentIndex`. We subtract this number from `currentIndex` to get the new index. The second method simply calls the first method for both the source and target nodes of the edges.

Listing 4 Node reindexing methods.

```
// === NODE REINDEXING ===

class Simulation {
  ...

  // Reindexes a single node
  reIndexNode(currentIndex, unlinkedIndexes)

  // Reindexes the source and target nodes of the links
  reIndexLinkNodes(links, unlinkedIndexes)

  ...
}
```

The `spreadInfection` method is the core of the simulation. We iterate over the edges of infected nodes and try to infect the neighbouring nodes. To not spread the virus too quickly, we must create a new array for the newly infected nodes and exclude them from the current spreading. After the loop is finished, we update the links of the newly infected nodes, the `infectedAmounts` array for the chart, and reheat the simulation to force it to tick.

Listing 5 Virus spreading method.

```
// === VIRUS SPREADING ===

class Simulation {
  ...

  drawSimulation(nodeCount, infectedPercentage) {
    ...

    // Spreads the virus
    function spreadInfection(intervalID, probabilities,
      spreadRate)
  }
  ...
}
```

The `Chart` class is used to create the chart. As it has no methods other than ones related to visualizing the chart, we will leave the description of this class for the next section.

Listing 6 Chart representation.

```
// === CHART ===

// Represents the chart
class Chart {
  ...
}
```

The `Data` class stores data for the simulation and handles user input. The constructor sets the default values of the simulation, creates the `Simulation` instance, and binds events to the DOM elements from the `index.html` file. The most notable event listener is the one for the start button. It schedules the repeated execution of the `spreadInfection` method every 1000 ms. The rest of the event listeners call simple methods of the `Data` class that change selections in the user interface, disable or enable elements, etc.

Listing 7 Class for data and user input.

```
// === DATA AND USER INPUT ===

// Represents the data and user input
class Data {
  // Initializes the data, simulation,
  // and binds events to the DOM elements
  constructor()

  ...
}
```

3.2.2 D3 for Visualization

We use D3 to create the graph and the chart for the simulation. The visualization part of the code is mostly taken from the examples provided on the D3 website and modified to fit our needs.

The visual representation of the graph is created by the `drawSimulation` method of the `Simulation` class, which we saw in listing 5. It pertains mostly to the visualization, which is why we have not discussed it in the previous subsection.

We first create a force simulation out of our nodes and edges. This includes creating the force that pushes the nodes with stranger connections further away. An SVG container is then created, and the nodes and edges are appended to it with their respective attributes visualized, such as color or width. Functions for dragging and ticking are also located in this method.

Listing 8 Simulation visualization.

```
// === SIMULATION VISUALIZATION ===

drawSimulation(nodeCount, infectedPercentage) {
  ...

  // Create a simulation with several forces
  const simulation = d3f.forceSimulation(nodes)
    .force("link"), d3f.forceLink(links)
    ...

  // Create an SVG container
  const svg = d3sel.create("svg")
    .attr('width', width)
    ...

  // Append the links to the SVG container
  // and set their attributes
  const link = svg.append("g")
    .selectAll()
    ...

  // Append the nodes to the SVG container
  // and set their attributes
  const node = svg.append("g")
    .selectAll()
    ...

  // Recalculates the positions of nodes and edges
  function ticked()

  // Functions for dragging nodes
  function dragstarted()
  function dragged()
  function dragended()
}
```

The SVG elements are appended to the page in the `index.html` file through the `createSVG` and `modifySVG` methods. The difference between the two is that

the former creates the initial random graph, while the latter removes the old graph and creates a new one based on the user input.

Listing 9 SVG handling methods.

```
// === SVG HANDLING ===

class Simulation {
  ...

  // Create SVG element with a random graph
  createSVG()

  // Replace SVG element with a graph based on user input
  modifySVG(nodeCount, infectedPercentage)

  ...
}
```

An example of how the graph looks can be seen in figure 3.2.

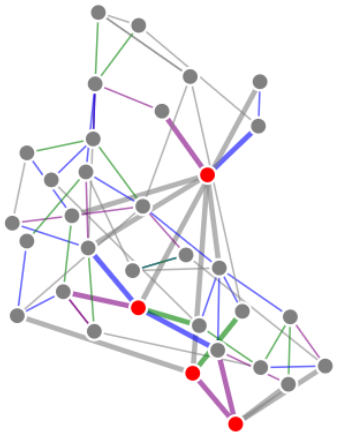

Figure 3.2 An example of a graph from the simulation.

The chart is created by the `drawChart` method of the `Chart` class. It uses the `infectedAmounts` array to create a line chart. We create an SVG element and append the x-axis and y-axis to it. We then create a line generator and append the generated line to the SVG element as well. The chart is updated every 5 ticks, as we want to show the sum of newly infected nodes from the last 5 ticks. The chart is appended to the page in the `index.html` file through the `createSVG` method.

Listing 10 Chart visualization.

```
// === CHART VISUALIZATION ===

class Chart {
  static drawChart(infectedAmounts) {
    // Create an SVG element
    const svg = d3sel.create("svg")
      .attr('width', width)
      ...

    // Add the x-axis and y-axis
    svg.append("g")
      ...
    avg.append("g")
      ...

    // Create a line generator
    const line = d3shp.line()
      ...

    // Append the line to the SVG element
    svg.append("path")
      ...
  }

  // Create SVG element with a chart
  static createSVG(infectedAmounts)
}
```

An example of how the chart looks can be seen in figure 3.3.

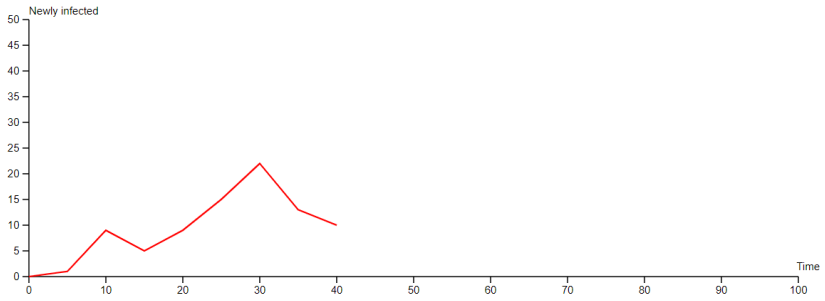

Figure 3.3 An example of a chart from the simulation.

3.2.3 Development Tools

We used Vite [35] to create the project and run it locally. It has many attractive features, such as NPM dependency resolving and pre-bundling. It also updates the page automatically after changes are made to the code, which makes the development process significantly faster. Vite supports both JavaScript and TypeScript out of the box. We contemplated using TypeScript, but decided against it due to problems further discussed in section 4.2.

The simulation was developed using Visual Studio Code as the main editor. We used Git for version control and GitLab for remote repository hosting.

Chapter 4

Results and Discussion

In this chapter, we first present the results of our work and how they match the goals we set in chapter 2. We then discuss the difficulties we encountered during the implementation and how we solved some of them. Finally, we suggest possible improvements and future work that could be done to enhance the simulation even further.

4.1 Results

Through a thorough analysis of interactive and educational simulations, we identified design goals that were necessary for creating a successful interactive pandemic simulation that encourages critical thinking. We then implemented a prototype by adhering to these goals.

The three main design goals we set for our implementation were empowering the user to experiment with the simulation, providing insight into the spreading of a pandemic, and doing so in a clear and concise manner. We believe that our implementation has met these goals. Let us discuss each of them separately.

Empowerment We have created a simulation that allows the user to experiment with various parameters, from the initial graph to the spreading of the virus. The user can try different scenarios, test their hypotheses, and question the author.

Explanation The user can see how the virus spreads through different relationships and the impact of restrictions on the spreading. An explanation of how each restriction affects the spreading is provided. This should build an understanding of how a pandemic spreads in a community and how different restrictions affect the spreading.

Understanding We have given the user enough options to experiment with the simulation without overwhelming them. The simulation is minimalistic and only shows the most important information. It is slow enough to follow the spreading of the virus, but fast enough to not bore the user. While it is not the most visually appealing simulation, it doesn't take away from the user's experience and could be fixed if we involved a designer in the process.

Note that the simulation is not perfect and is not meant to be a complete solution to the problem of pandemic simulations. It is a prototype that shows how future models can be made more socially beneficial and how they can be used to educate the public.

4.2 Difficulties with D3

Unexpected difficulties arose when using the D3 library in our implementation. We have encountered several very specific technical problems that are part of a larger issue with the library's documentation. The documentation is spread across several versions, making it difficult and sometimes impossible to find relevant information. We will discuss some of the issues to illustrate the problems we faced along with the solutions we found.

We started by playing around with one of the many examples available on their website. Our intention was to get hands on experience with the library and to understand how it works. We quickly realized that simply copying and pasting the code would not be enough. It appears that the examples don't show the entirety of the code, and some parts are left out. We had to figure out what to change ourselves. The solution was rather simple, as we only needed to wrap the code in a function instead of assigning it to a variable. However, this was not immediately obvious, and we believe it would be beneficial to have examples that don't need to be modified locally in order to work.

JavaScript has some drawbacks that preprocessors aim to solve, TypeScript being one of the best options [36] [37]. That is why after we gained a basic understanding of how D3 works, we decided to explore the possibility of using it with TypeScript. We didn't find any official documentation on this. Once we installed the necessary packages and rewrote the code in TypeScript, errors from within the D3 library started appearing upon compilation. For example, the library mutated the types of nodes and edges in its functions. Such errors were solved by adding `//@ts-ignore` comments to the code, but it was not an ideal solution. Due to the tedious nature of solving all the errors, we decided to revert back to JavaScript.

Another issue we encountered was the tick handler. Instinctively, we assumed

that the tick handler `simulation.on('tick', ticked)` would be called every time the simulation updated. However, this was not the case. It was only called if the nodes in the simulation were moving. This caused the simulation to stop updating after a while, even though the nodes were still internally changing. We tested this by waiting for the simulation to stop updating, letting several spreading cycles pass, and then moving the nodes manually. The simulation updated and most of the nodes were infected. This would be impossible in a single spreading cycle, so it confirmed our suspicion. We solved this by having the `setInterval()` function call the spreading function. At the end of the spreading function, we reheat the simulation by calling `simulation.restart()`. This forces the simulation to tick and update.

4.3 Future Work

The work presented in this thesis has many areas that could be improved upon.

Generating Relationships The type of the edges is fully randomized as of now. Assigning a probability to each relationship type would allow for a more realistic model of a community.

States The current model only supports two states - infected and not infected. This could be expanded upon to include more states depending on the direction we want to take. Examples of possible states include recovered, dead, and vaccinated. While this would allow for a more detailed simulation, it would also make the graph harder to read during a running simulation. It would require a more complex visualization system to display the simulation in a clear and concise manner.

Graph Visualization A possible solution to the problem above and a general improvement to the current system would be to implement a visualization system where the user can choose by which parameters they want to group the nodes by. Grouping them by their connections would allow for interesting insights into how different restrictions affect different relationships, while grouping them by their state would make the speed of the spread more apparent.

Interactivity Various interactive features could be added to the simulation. This includes more customability of the initial graph, adding restrictions, and the ability to change how strongly a restriction affects the spreading probabilities. This would encourage critical thinking even more, as the user would have many more tools to experiment with.

User Interface The current user interface is very basic. A more polished interface would make the simulation more attractive to a wider audience. This mostly contains visual improvements, such as better color schemes and more intuitive controls.

Conclusion

The goal of this thesis was to create an interactive pandemic simulation that encourages critical thinking. We believe that through our design, we have achieved this goal.

We have successfully applied the knowledge from our analysis to design a simulation and implemented a prototype that fulfills all of our design goals. The final simulation is interactive, engaging, and encourages the reader to think critically about the spread of a virus. While our implementation isn't fully polished, it serves as a proof of concept for our design and a starting point for future improvements.

It should be noted that the simulation is not a replacement for accurate epidemiological models. It is not meant to be used for making decisions about public policies or predicting the spread of a virus. Instead, it is designed to be a tool for education and critical thinking. We believe that based on our design, a simulation that journalists could use to explain the spread of a virus to the general public could be created.

In conclusion, we believe that our simulation design is a success. We hope that our work will serve as a step towards creating more socially beneficial models that help people understand complex topics such as the spread of viruses.

Bibliography

[1] Eric Winsberg. "Computer simulations in science". In: (2013).

[2] Madhav Marathe and Anil Kumar S Vullikanti. "Computational epidemiology". In: *Communications of the ACM* 56.7 (2013), pp. 88–96.

[3] Bret Victor. *Explorable Explanations*. 2011. URL: https://worrydream.com/ExplorableExplanations/.

[4] Harry Stevens. *Why outbreaks like coronavirus spread exponentially, and how to "flatten the curve"*. 2020. URL: https://www.washingtonpost.com/graphics/2020/world/corona-simulator/.

[5] Tony Chu and Stephanie Yee. *Making sense of COVID19 through simulations*. 2020. URL: http://www.r2d3.us/covid-19/.

[6] U. Wilensky. *NetLogo Fire model*. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. 1997. URL: http://ccl.northwestern.edu/netlogo/models/Fire.

[7] Michael Mark Woolfson, MM Woolfson, and Geoffrey J Pert. *An introduction to computer simulation*. Oxford University Press, USA, 1999.

[8] David Gibson, Clark Aldrich, and Marc Prensky. *Games and simulations in online learning: research and development frameworks*. IGI Global, 2006.

[9] J Peter Kincaid et al. "Simulation in education and training". In: *Applied system simulation: methodologies and applications* (2003), pp. 437–456.

[10] Dimitrios Vlachopoulos and Agoritsa Makri. "The effect of games and simulations on higher education: a systematic literature review". In: *International Journal of Educational Technology in Higher Education* 14 (2017), pp. 1–33.

[11] Sherry Turkle. *Simulation and its discontents*. MIT press, 2009.

[12] Jeffrey Johnson. “The “can you trust it?” Problem of simulation science in the design of socio-technical systems”. In: *Complexity* 6.2 (2000), pp. 34–40. DOI: https://doi.org/10.1002/cplx.1017. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cplx.1017. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cplx.1017.

[13] Lev Manovich. “What is visualization?” In: *paj: The Journal of the Initiative for Digital Humanities, Media, and Culture* 2.1 (2010).

[14] FT Visual & Data Journalism team. *Coronavirus tracked: see how your country compares*. 2022. URL: https://ig.ft.com/coronavirus-chart.

[15] Lily W Ge, Yuan Cui, and Matthew Kay. “Calvi: Critical thinking assessment for literacy in visualizations”. In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 2023, pp. 1–18.

[16] Bret Victor. *Ten Brighter Ideas? An Explorable Explanation*. 2010. URL: https://worrydream.com/TenBrighterIdeas/.

[17] May Yong, Thomas Knowles, and Tomáš Petříček. *Accounting for democracy*. 2024. URL: http://turing.thegamma.net/expenditure.

[18] Daniele Vernon-Bido, Andrew Collins, and John Sokolowski. “Effective visualization in modeling & simulation”. In: *Proceedings of the 48th annual simulation symposium*. 2015, pp. 33–40.

[19] Jasna Kuljis, Ray J Paul, and Chaomei Chen. “Visualization and simulation: Two sides of the same coin?” In: *Simulation* 77.3-4 (2001), pp. 141–152.

[20] U. Wilensky. *NetLogo*. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. 1999. URL: http://ccl.northwestern.edu/netlogo/.

[21] Owen Densmore. *AgentScript*. 2024. URL: https://agentscript.org/.

[22] Sean Luke et al. “Mason: A multiagent simulation environment”. In: *Simulation* 81.7 (2005), pp. 517–527.

[23] Seth Tisue and Uri Wilensky. “Netlogo: A simple environment for modeling complexity”. In: *International conference on complex systems*. Vol. 21. Citeseer. 2004, pp. 16–21.

[24] Mike Bostock and contributors. *D3.js*. 2024. URL: https://d3js.org/.

[25] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. “D^3 data-driven documents”. In: *IEEE transactions on visualization and computer graphics* 17.12 (2011), pp. 2301–2309.

[26] Scott Murray. *Interactive data visualization for the web: an introduction to designing with D3*. " O’Reilly Media, Inc.", 2017.

[27] Christian Swinehart. *Arbor.js*. 2024. URL: http://arborjs.org/.

[28] Alexis Jacomy and contributors. *Sigma.js*. 2024. URL: http://sigmajs.org/.

[29] Jakob Heuser. *El Grapho*. 2024. URL: http://elgrapho.com/.

[30] Richard M Burton and Børge Obel. "The challenge of validation and docking". In: *Proceedings of the Workshop on Agent Simulation: Applications, Models, and Tools*. Citeseer. 1999, pp. 216–221.

[31] Andrew J Collins, D'An Knowles-Ball, and Julia Romberger. "Simulation visualization issues for users and customers". In: *Proceedings of the 48th Annual Simulation Symposium*. 2015, pp. 17–24.

[32] Neil M Ferguson et al. "Impact of non-pharmaceutical interventions (NPIs) to reduce COVID-19 mortality and healthcare demand. Imperial College COVID-19 Response Team". In: *Imperial College COVID-19 Response Team* 20.10.25561 (2020), p. 77482.

[33] Henry Story. *Code, Models and Covid-19*. 2020. URL: https://medium.com/@bblfish/open-source-and-covid-19-models-5e638f785514.

[34] Averill M Law, W David Kelton, and W David Kelton. *Simulation modeling and analysis*. Vol. 3. Mcgraw-hill New York, 2007.

[35] Evan You and contributors. *Vite*. 2024. URL: https://vitejs.dev/.

[36] Manuel Mertl. "Comparision and evaluation of JavaScript preprocessing languages". PhD thesis. Wien, 2016.

[37] Justus Bogner and Manuel Merkel. "To Type or Not to Type? A Systematic Comparison of the Software Quality of Javascript and Typescript Applications on GitHub". In: *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022, pp. 658–669.

Appendix A

Using the Simulation

To compile and run the simulation locally, you only need to have Node.js installed. You can download it from `https://nodejs.org/en/download/`. The rest is handled by the package manager npm, which comes with Node.js.

Dependencies

The simulation uses several D3 packages, Vite, and JSDoc. You can easily install them by running the following command in the root directory of the project:

```
npm install
```

Generating Documentation

The simulation uses JSDoc to generate documentation. You can do so by running the following command in the root directory of the project:

```
npx jsdoc src/main.js
```

Running the Simulation

Once you have all the dependencies installed, you can run the simulation by running the following command in the root directory of the project:

```
npx vite
```

This will start a local server. The default address is `http://localhost:5173/`. If the port is already in use, Vite will automatically choose the next available port. The correct address will be printed in the terminal. You can access the simulation by opening this address in your browser.

Navigating the Simulation

The simulation has an intuitive user interface. It includes an explanation of the representation of the simulation, as well as the controls. You can configure the parameters of the simulation and run it by pressing the start button. The simulation will then run until you stop it or until all the nodes are infected.