TALLINN UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

Uku Markus Tammet 176475 IAPM

SIMULUS: A CRISMA-INSPIRED SIMULATION FRAMEWORK FOR WEB BROWSERS

Master’s thesis

Supervisor: Tanel Tammet

PhD

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Uku Markus Tammet 176475 IAPM

SIMULUS: CRISMAST INSPIREERITUD SIMULATSIOONIRAAMISTIK VEEBIBRAUSERITELE

Magistritöö

Juhendaja: Tanel Tammet

PhD

Tallinn 2019

Author’s declaration of originality

Abstract

The subject of this thesis is crisis simulation, based on agent-oriented modelling. We design and implement a framework for building simple crisis simulations, inspired by the CRISMA project. Instead of focusing on flexibility or completeness, we focus on ease of use and low barriers of entry. The goal is to make the results of state-of-the-art research usable by people interested in building simple practical crisis simulations without requiring significant technical and organizational resources. The main result of the thesis - the implementation of the SIMULUS framework - is novel in the sense that the simulation developer has no need for any other IT resources except the browser: the whole system is deployed as a static Javascript-based web application without any external databases or services. To achieve this, we utilize a number of modern browser technologies. Finally, we implement a simple reference simulation application of mass casualties on top of this framework.

The thesis is written in English, contains 49 pages of text, 6 chapters and 11 figures.

Annotatsioon

SIMULUS: CRISMAst inspireeritud simulatsiooniraamistik veebibrauseritele

Käesoleva magistritöö sisuks on agentorienteeritud modelleerimisel põhinevad kriisisimulatsioonid. Oleme disaininud ja realiseerinud tarkvara-raamistiku lihtsate kriisisimulatsioonide ehitamiseks. Raamistiku ideoloogia ja arhitektuur on inspireeritud CRISMA projektist ning tema fookuses ei ole mitte paindlik ja rikas funktsionaalsus, vaid kergus ja kasutajasõbralikkus. Töö eesmärgiks on pakkuda kriisimulatsioonide ehitamisest huvitatud inimeste jaoks tipptasemel uurimistulemustele baseeruvad ehitusblokid, mille kasutamine ei nõua suuri tehnoloogilisi ja administratiivseid ressursse. Loodud raamistik SIMULUS on ehitatud ainult veebitehnoloogiatega, ning selle abil ehitatud rakendused on staatilised javascripti-põhised veebirakendused, mis ei vaja väliseid andmebaase, servereid ega muid IT-teenuseid. Vajaliku funktsionaalsuse saavutamiseks brauseris kasutasime modernseid veebitehnoloogiaid. Lisaks ehitasime oma raamistikku kasutades lihtsa kriisisimulatsiooni näidisrakenduse.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 49 leheküljel, 6 peatükki ja 11 joonist.

List of abbreviations and terms

API	Application programming interface, a clearly defined perimeter of communication in software in the form of protocols or definition.
CRISMA	Modelling crisis management for improved action and preparedness, an EU integration project.
ECMAscript	The scripting-language specification that JavaScript implements.
ICMS	Integrated crisis modelling system, a sub-project of CRISMA.
IndexedDB	A client-side database integrated into most browsers.
MIT license	A permissive open-source license.
NPM	Node package manager, a package manager for the JavaScript programming language.
OOI	Object of interest, an entity in a simulated world that constitutes a part of the world state.
OOP	Object-oriented programming, a ubiquitous programming paradigm.
Rust	A systems programming language focused on safety.
UUID	Universally unique identifier, a type of identifier used in software.
Web Worker	A browser standard, allowing developers to run code in new threads in browsers.

Table of contents

List of figures

Introduction

The goal of this thesis is implementing modern approaches to agent-oriented crisis simulation using browser-only technologies, in order to maximise ease of use and lower the barrier of entry. In short, we want to make crisis-simulation-building approachable both by university courses and small and medium public sector organisations.

The reader could ask, ”why crisis simulations”? While modern technology is breaking down traditional barriers for movement and communication, resource coordination has become more efficient than ever. This is obviously important for public institutions and critical systems of infrastructure, especially in times of crisis. The relevant parties often implement various contingency plans to increase their preparedness for different crisis situations. Planning for these contingencies involves building a formal or informal model of the crisis situation and the objects of interest and behaviours affecting it. If possible, these are based on existing data and existing historical situations exemplifying an instance of the contingency being planned for. Strategies for optimal resource coordination and behaviour during such crises are then built using these models. However, depending on the stuation being modelled, testing their efficacy, or even having the data to build a model can be difficult. One practical method for validating these strategies is to formalise the model – for example, as an agent-based model – and utilise simulation software to test said strategies [3].

The “Modelling crisis management for improved action and preparedness” or CRISMA project was was an integration project financed under the Security theme of the Cooperation Programme of the 7th Framework Programme of the European Commission [4]. Among other work, the project produced a conceptual and technological framework that can be used for visualising, simulating and evaluating different mitigation strategies for crisis situations [2]. The framework has been utilised in building various pilot projects with successful results.

The architecture created by the CRISMA project is built utilising previous work and relies on a complex distributed service-oriented architecture (SOA). For the specific goals of the CRISMA project, this is a necessary approach, as some of the key system requirements are interoperability with existing applications, models and data [5].

Like CRISMA, most proposed architectures require substantial investment into the design, architecture, infrastructure and maintenance of simulation tools. This restricts this valuable research to businesses and agencies with enough resources to utilise it. In addition, the resource requirements may sway parties otherwise interested in simulation systems from using this type of modelling. If we are able to design a more convenient system by taking coordinated tradeoffs, we may be able to have more parties benefiting from this research.

Thus the goal of this thesis: design and implement a simple software framework / platform for would-be crisis simulation designers, based purely on browser technologies while employing the conceptual results and ideas of the current state-of-the-art projects, in particular, CRISMA.

The SIMULUS software is open source under the MIT licence. The source code, a demonstration application and documentation is available at https://github.com/tankenstein/simulus

1 Background

In this chapter, we will give a brief overview of crisis simulation, how agent-oriented modelling applies to it and how recent changes in the browser platform can impact this.

1.1 Crisis simulation

In times of crisis, the better prepared we are, the better the outcome. However, preparing for different crises, especially in today's complicated world, can be difficult. On one hand, you often don't know how all the variables at play would affect each other. Although you could rely on historical examples of the contingency you're planning for, you might be facing a situation that has not been encountered before. For example - what if there was a modern chemical attack in a large sprawling metropolis. How would you know how to prioritise resources, or even who and where to evacuate first? How will your normal evacuation procedures be affected by a deadly chemical contaminant? To help in resolving these questions, of which there are many, we can both run software simulations and play out simulated crisis situations with actual people in a ”game” format. However - planning isn't the only thing crisis simulations can help with. As Kleiboer puts it [6], we can divide crisis simulations goals into research or educational goals. The former would be for activities such as planning contingency methods for crisis situations, testing those methods or for designing a decision support system. The latter would be for training and selecting personnel [6]. As the authors of the CRISMA crisis simulation project's main book put it [2]: ”It can be almost impossible to completely understand complex systems, but it is possible to get used to them and learn how they behave in certain conditions. In order to do so, it is important to train yourself by “playing” with the system”.

Crisis simulation has its roots in warfare [6]. In modern times, crisis simulation involves either software running a simulation, people simulating a crisis situation, or having people and software simulating and pracicing a scenario in tandem. Historically, crisis simulation has only meant so-called ”games”, where different scenarios were practised by people [6]. Nowadays, crisis simulation has evolved past warfare and made its way into many different types of organisations, including in the private sector [6].

1.2 Agent-oriented modeling in crisis simulation

While the field of modeling and simulation is a large emerging field, producing many ways to build and model simulations to accomplish some specific analysis, in this paper we focus more specifically on agent-oriented modeling. Building simulations with agent-oriented modeling differs from other modeling and simulation methods in that it is bottom-up rather than top-down, meaning that behaviour is not enforced upon components, but rather the behavour of the system is emergent, from the defined behaviour, relationship and interactions of individual components [7]. When we look at crisis situations in the world, we find that they mirror this approach of modeling quite well, although developing computer simulations of real-world situations is still difficult [8]. Large crisis situations often have a multitude of different actors, whose interactions and decisions affect the outcome in significant ways.

To illustrate this, let’s bring an example of a crisis situation. There is a fire in a high-rise building in a town. There is multitude of different actors of different roles involved in such an event: general coordinators, dispatchers, emergency services personell, bystanders and residents of the tower. To see how effective a particular strategy to lessen the effects of such a crisis is, you would have to not only know particulars about the fire, but particulars about the agents involved and their interactions. For example, the likelihood of being able to stop the fire depends on the arrival time of the first responders. That depends on the speed and location of the first responders, the effectiveness of communication between dispatchers, bystanders and general coordinators. Here you can see how an ultimately simple measurement results from a complex chain of relationships and dependencies.

When modeling crisis situations, it’s vital to discern to what goal and to which user your model and simulation is catered to. Referring back to the five different main use cases Kleiboer highlights in their work [6], these different use cases can have vastly different technological requirements, but as the underlying domain we are modeling is the same, agent-oriented approach can theoretically work for all of them.

1.3 Recent changes in web-based simulation

As we can often effectively model crisis situations using agent-oriented modeling, we can theoretically select one of many available agent-oriented simulation tools to run our crisis simulations. In ”Agent Based Modelling and Simulation tools: A review of the state-of-art software”, the authors give a thorough overview of 83 such toolkits, categorising them using a number of different criteria, including but not limited to their platform, license, difficulty of model development, computational scalability, an a assessment of the domains the tool is suitable for and implementation types of agents [9]. For crisis simulation, we often require the ability to model systems with GIS data. Applying this criteria to the reviewed tools, we are left with the following tools [9]:

- Cormas
- Envision
- GAMA (2D/3D)
- Insight Maker
- MATSim
- OBEUS
- Pandora
- Repast-J/Repast-3
- Repast HPC
- Repast Simphony (2D/3D)
- SOARS
- TerraME

In our thesis we are specifically interested in the viability of using the browser as a platform for our crisis simulation tool. J. Byrne, C. Heavey and P.J. Byrne gave a comprehensive overview of the history, evolution and comparative advantages / disadvantages of web-based simulation in 2010 [10]. However, since 2010 the browser environment has markedly evolved, relying on native browser technology, mainly thanks to advances in the ECMAscript toolset and specification. This, along with a track record of security vulnerabilities, has rendered traditional java applets and alternative multimedia approaches (e.g. flash) largely obsolete.

The main different approaches for web-based simulation listed in the Byrne, Heavy and Byrne paper shift various responsibilities between the server and the browser, making different tradeoffs along the way [10]. In this paper, we specifically focus on the ”Local simulation and visualisation” approach outlined in the aformentioned paper, which is described in the paper as follows: ”where both the simulation engine and visualisation components are downloaded seamlessly by the client to the user’s local computer, so that the graphical interface and the simulation engine coexist in the same environment (i.e. within the browser) shifting the responsibility for execution completely from the server to the client” [10]. The evolution of the toolset removes or minimizes a number of disadvantages of this approach noted in their paper. The graphical user interface limitation they outline has become largely obsolete. In fact, application builders now often prefer the web’s toolkit to native toolkits or java-based GUI toolkits. In addition, the performance of using a local simulation and visualisation system has improved, thanks in part to advances in the performance of the available javascript JIT-compilers and also to new technologies, such as Web Workers and SharedArrayBuffers, allowing developers to parallelise computation in browsers, not to mention just the average device getting more powerful.

One example of utilising modern browser technology to build a web-based simulation platform can be found in our previous list of potential tools from [9]. InsightMaker, detailed by Scott Fortmann-Roe in 2014 [1], provides sophisticated modelling and simulation capability, running entirely in a web browser, with a focus on accessibility for users. It also includes an analysis environment with different available graphs. This environment might not however be ideal for simulating and analysing crisis situations specifically, as it lacks some fundamental features such as visualising geographic locations and the ability to navigate between world states effectively. These capabilities are not built in on purpose, as the author intended to sacrifice some features for added accessibility to avoid software bloat [1]. These are features that may not be necessary for a general-purpose simulation tool.

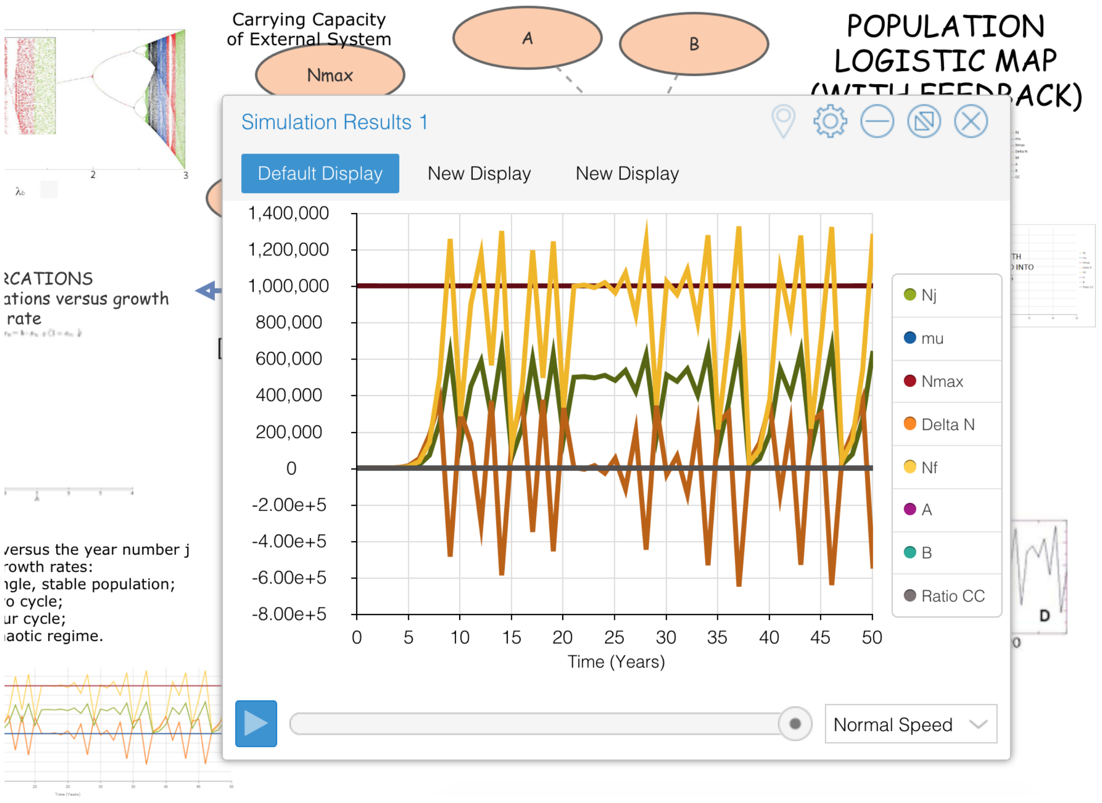


Figure 1: Insightmaker[1] analysis environment

2 CRISMA

2.1 CRISMA project

The “Modelling crisis management for improved action and preparedness” or CRISMA project was was an integration project financed under the Security theme of the Cooperation Programme of the 7th Framework Programme of the European Commission that ran from March 2012 to August 2015, where TalTech was one of the collaborating universities devoting personnel and capital to the project [4]. The goal of the project is ”improved crisis management in large-scale and complex crisis scenarios” [2]. Among other work, the project produced an architecture and technological framework implementing this architecture that can be used for visualising, simulating and evaluating different mitigation strategies for crisis situations [2], along with specifications and reference applications. The resulting framework is meant to address a wide range of use cases, but they highlight six specific ones for crisis simulation [11]:

- Long Term Planning
- Incident Evolvement
- Resource Planning
- Decision Training
- Exercise Support
- Strategic and Operational Planning

2.2 CRISMA framework

This section provides a brief overview of what the CRISMA framework provides, mainly sourced from the CRISMA ICMS architecture document v2 [5].

The CRISMA framework provides technological underpinnings to build crisis simulation applications based on the CRISMA architecture. More specifically, to quote the main public CRISMA book [2], it provides:

- Rules and guidelines for the development of CRISMA applications
- A middleware for the integration of CRISMA building blocks and legacy components
- A generic core control and communication information model (Core CCIM) that is usable by any CRISMA application
- Software realisations of all building blocks specified in the architecture (”software components”), and
- A set of reference applications which may act as a blueprint for the development of concrete CRISMA applications.

It is based on a distributed paradigm, where different software services provide services to each other, in a service-oriented architecture [2]. To build a fully-realised CRISMA application, it is meant that you integrate selected CRISMA-defined services and your own legacy application services together, by combining them with provided CRISMA building blocks [12]. You then also need a UI that calls these services to make use of them, and the project provides common building blocks for certain UI elements as well.

A central part of the framework is the Integrated Crisis Management Middleware, or ICMM for short. It integrates multiple pieces of the whole application together, like legacy data api-s and simulation runners. It is also is responsible for providing a central repository of core simulation data, like world states and indicators [5].

The building blocks CRISMA provides are categorized into three main groups. These groups are the following [5]:

- Infrastructure - core CRISMA components that other BBs rely on, like Pub/Sub or the ICMM.
- Integration - these allow you to integrate your own models and indicators on top of CRISMA infrastructure.
- User interaction - various core UI components (implemented for browsers) that enable common crisis imulation interactions, like browsing world states, that you can slot into your custom UI if your data model conforms to CRISMA standards.

2.3 CRISMA framework data model

In this section we explain some of the critical types of data that the CRISMA framework uses to model real-life scenarios [5].

Scenario

Usually, when using a CRISMA application, you run a simulation in the context of a ”scenario”. A ”scenario” contains a self-contained simulation, that is to say, all other properties used in the simulation are specific to that instance of a scenario.

World state

In these scenarios, the world is modeled on a timeline of states, where a single state contains the entire state of the simulated world at the time, or a pointer to another world state [5]. These states can branch from each other, allowing a user to try different strategies or different problems, while starting from a common state. States are immutable, when you want to change something, you create a new state from your modifications or from running a simulation [5].

Object of interest

Since we are simulating agents, some of the most important pieces of data contained in a world state are representations of the entities in the world. An object of interest (OOI) is some distinct entity or actor in the world we are modelling. For example, a single patient or a specific rescuer. They consist of some meta-information (identifier, name and the like), but more critically, they have a specific object of interest type (OOI type) that they conform to, that defines their behaviour and possible properties. In addition, they have a defined geometry that ties them into a location or area of the simulated world.

Object of interest type

An object of interest type (OOI Type) defines a type of entities, defining how instances of this type behave and what properties (besides the common ones) they posess. For example, this could be a ”patient” type, defining properties such as ”condition” and ”frailty”. Usually behaviour is assigned to specific groups instances of these types. These types can also extend each other, for example, a Doctor type might have a Person type as its parent, where instances of doctors would inherit all the properties of a person.

Object of interest property

Object of interest properties (OOI properties) define what properties entities of that type have. These properties define the state of the entity in the world. For example, a single patient, of the ”Patient” type, can have a ”condition”, that can deteriorate over time. The property can be of different actual data types.

Geometry

This is an object representing the location or space that something occupies, usually in 2D space. This is flexible, it can be a single point, a line, or any polygon.

Indicator

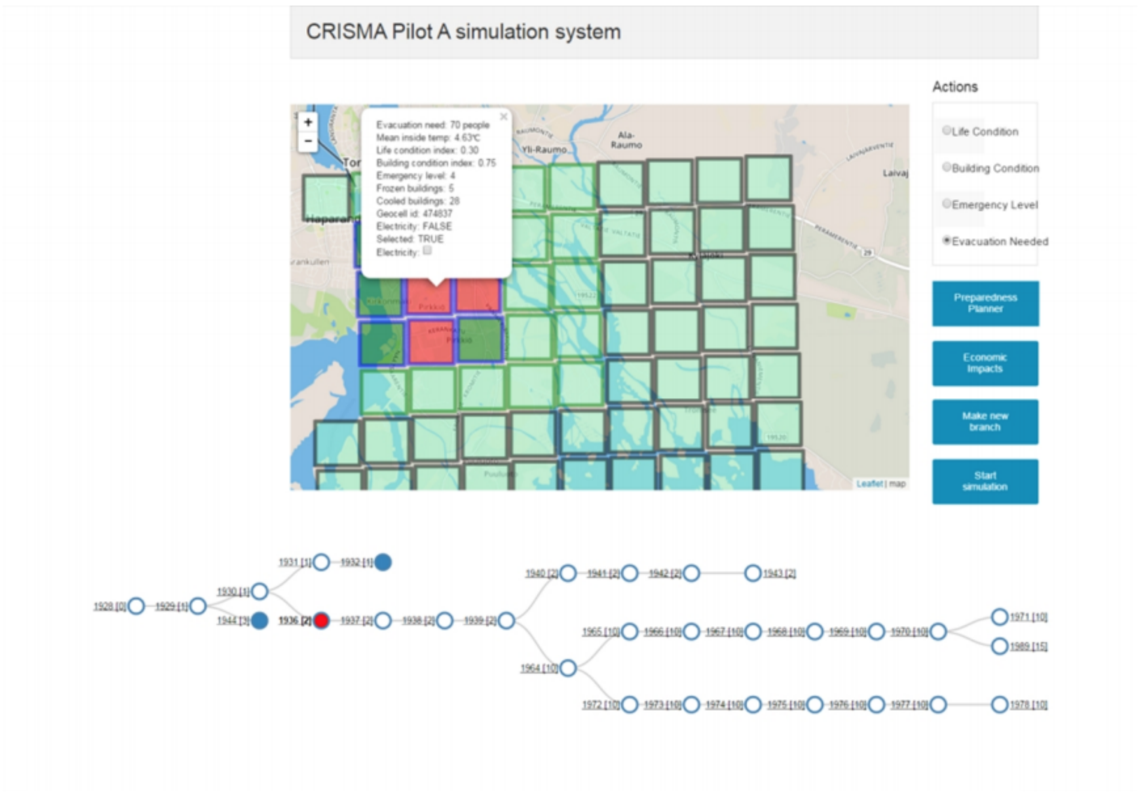
In addition to entities, the world state also contains information about the current indicators, which are functions that run on a given world state that provide some analysis on the state. For example, in a mass-casualty incident, it is vital to see how the current casualties are doing - in which states they are. For this, you can define an indicator function, that will group your ”patient” entities into groups, and output the counts of those groups.

2.4 CRISMA pilot projects

As previously discussed, the CRISMA project provides reference applications for different domains. These reference applications are collections of CRISMA building blocks, but meant so you would add your own data into them. During the project, the framework was utilised in building various pilot projects on top of these reference applications with successful results. Five pilot projects were selected and developed. These reflected the following types of crisis simulations [2]:

- Extreme winter weather crisis in the north of Europe
- Coastal submersion defence strategies for the Charente-Maritime county
- Earthquake and forest fire application
- Accidental spillage from a container at a large city port
- Mass-casualty incident

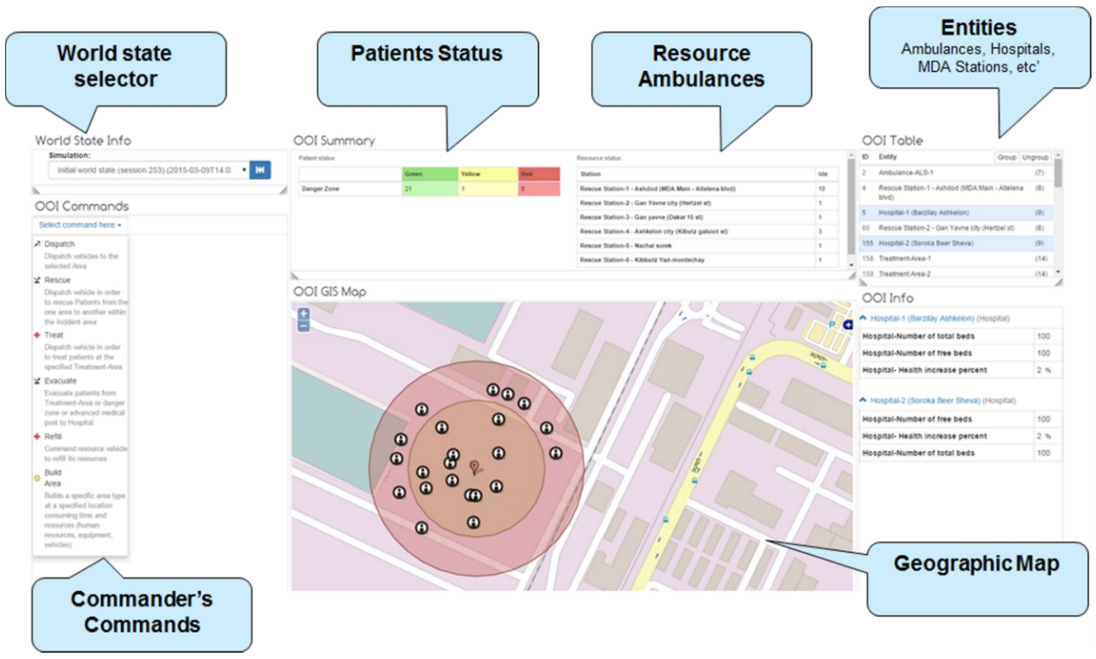
These applications provided solutions for different use-cases, with different technical requirements and different problem domains, but utilising the same core CRISMA components and building blocks.



Evacuation need: 70 people
Mean inside temp: 4.63℃
Life condition index: 0.30
Building condition index: 0.75
Emergency level: 4
Frozen buildings: 5
Cooled buildings: 28
Geocell id: 474837
Electricity: FALSE
Selected: TRUE
Electricity: ☐

Figure 2: Evacuation need visualisation (from [2])

The above figure shows a view of the “extreme winter weather crisis” CRISMA pilot application. Here, a user can analyse how people in different affected areas are currently doing and decide how to use the evacuation measures at their disposal [2].



Simulation: Initial world state (session 253) (2015-03-09T14:0

Select command here

Dispatch
Dispatch vehicles to the selected Area

Rescue
Dispatch vehicle in order to rescue Patients from the one area to another within the incident area

Treat
Dispatch vehicle in order to treat patients at the specified Treatment-Area

Evacuate
Evacuate patients from Treatment-Area or danger zone or advanced medical post to Hospital

Refill
Command resource vehicle to refill its resources

Build Area
Builds a specific area type at a specified location consuming time and resources (human resources, equipment, vehicles)

Patient status

	Green	Yellow	Red
Danger Zone	21	1	0

Resource status

Station	Idle
Rescue Station-1 - Ashdod (MDA Main - Altelena blvd)	10
Rescue Station-2 - Gan Yavne city (Hertzel st)	1
Rescue Station-3 - Gan yavne (Dakar 15 st)	1
Rescue Station-4 - Ashkelon city (Kibutz galuiot st)	3
Rescue Station-5 - Nachal sorek	1
Rescue Station-6 - Kibbutz Yad-mordechay	1

ID	Entity	Group Ungroup
2	Ambulance-ALS-1	(7)
4	Rescue Station-1 - Ashdod (MDA Main - Altelena blvd)	(8)
5	Hospital-1 (Barzilay Ashkelon)	(9)
60	Rescue Station-2 - Gan Yavne city (Hertzel st)	(8)
155	Hospital-2 (Soroka Beer Sheva)	(9)
158	Treatment-Area-1	(14)
159	Treatment Area-2	(14)

Hospital-1 (Barzilay Ashkelon) (Hospital)

Hospital-Number of total beds	100
Hospital-Number of free beds	100
Hospital- Health increase percent	2 %

Hospital-2 (Soroka Beer Sheva) (Hospital)

Hospital-Number of free beds	100
Hospital- Health increase percent	2 %
Hospital-Number of total beds	100

Figure 3: Spillage application training view (from [2])

The above figure shows a view of the "Accidental spillage from a container at a large city port" CRISMA pilot application which can be used to train crisis commanders by reacting to a simulated spillage situation [2].

2.5 On complexity of crisis simulation

Obviously, the CRISMA framework is a very complex collection of services, standards and building blocks. A reason it is so complex can be attributed to the fact that there is currently no obvious way to abstract crisis simulation into something generic while retaining enough customisability. When building simulations, we have to describe behaviour of a system in concrete terms for a computer to execute. A really efficient tool for this is computer code. This means that to utilise crisis simulation effectively, one usually has to either build an entire application or to at least build their own model for their specific simulation, unless their usecase is already covered well enough by existing applications. What is more, crisis simulation can be computationally very complex, with different simulations having very different technical requirements. The data model that might work for a mass-casualty event ambulance response simulation might be really inefficient for a river flooding simulation.

All of these requirements for domain and technical flexibility mean that if we want to build a toolkit to help organisations run, and to run better simulations we can't just provide a single configurable application. What we do instead is to provide them with a loose collection of resources, tools and standards that they can then combine with their own facilities in order to build a custom simulation application. As the authors put it in the main CRISMA book [2], ”The CRISMA Framework architecture is kept as simple as possible, and as complex as necessary, in order to address the wide range of use cases, integrate legacy systems, adapt to various security constraints and integrate the diverse range of hazard, impact and risk models.”

3 Simulus architecture and technology

In this chapter we describe the design priorities and architecture of our CRISMA-inspired web crisis simulation toolkit, but first we will reiterate our hypothesis. In the effort to cover all possible current and future use cases for crisis simulation and the different technological requirements of different simulations, the CRISMA framework has opted for a complex service-oriented architecture, that allows the created toolkit to adapt to most needs. However, this architecture brings with it a number of downsides. It is a relatively complex endeavour to build and set up a crisis simulation application using the framework. To mitigate this, the CRISMA authors have also included a set of reference applications that you can build your own application on top of, requiring less work to figure out the wiring. In addition to the startup cost, such an architecture requires some infrastructure and requires one to maintain said infrastructure, which can bring with it some personnel and capital requirements.

What this means is that the CRISMA framework is geared towards larger organisations with enough resources to spare to overcome the setup and maintenance. The highest need for crisis simulation comes from these organisations, so it makes perfect sense to gear the framework towards them. However, we believe there is value in making a simple crisis simulation tool accessible for the layman. This would encourage and allow more people to use this useful research methodology.

3.1 Priorities

The goal of our toolkit is to make it easy to run and maintain simple crisis simulations. This means that in contrast to some other crisis simulation toolkits that focus on completeness and flexibility, our focus should be accessibility. Thus, in our case we are trading off some flexibility and use cases for ease of use. However, while simple, simulations created using our toolkit should still be useful for specific use cases and contexts that we define. In order to be useful, when building a simulation using our toolkit, one should be able to utilise the resulting application in order to:

- Familiarise oneself with the behaviour of a complex system in certain conditions, by ”playing” with the system [2].
- Plan contingency methods for certain crisis scenarios
- Test the effectiveness of planned contingency methods

As discussed previously, it is ineffectual to try and abstract the behavioural and modelling aspect of crisis simulation. Thus, we need to build a system where people can integrate their own behavioural code and domain entities into our toolkit. The parts that our toolkit should provide are things that are relatively common between crisis simulation. Looking at CRISMA and the different CRISMA pilot projects, we’ve identified the following facilities our framework should provide:

- Most of the common UI - managing world states, entities and the like
- Utilities for analysis
- Persistence of data
- Simple integration of custom data and behavioural model into our framework

A large deviation we decided to make from the CRISMA framework is that our framework will be the top-level software actor in the entire system, responsible for coordinating all the other components. In CRISMA, the framework is distributed as a set of tools you integrate with your systems. In our system, it’s the opposite. You integrate your tools into our framework. This is effectively a tradeoff between flexibility and accessibility. Our choice requires less work on the user’s side, but is much less flexible.

3.2 Browser platform constraints

In order to minimize setup cost and maintenance cost we have decided to build our platform entirely as a browser application, where running it is only a matter of hosting static files somewhere, or accessing them on your own machine. In addition, the modern browser is an excellent platform to build user interfaces on, in fact, CRISMA applications use browsers for their user interface, and the interaction building blocks supplied by CRISMA are built using browser technologies. While the usage of the JavaScript language for this sort of application may seem unorthodox to some, the language is being used extensively in the scientific world, for example, by the James Webb Space Telescope's space operations software [13]. However, this approach has a number of constraints we have to keep in mind:

- Limited computational capacity - since all the code is running in the user's browser, we cannot scale across multiple machines. In addition, JavaScript, while relatively fast, is still slower for raw computation than languages more suited for such tasks, like C or Rust. However, modern browser technologies like Web Workers can help us parallelise our JavaScript.

- Loss of integration flexibility - since we do not want to create dependencies on other web services, integrating legacy data or systems cannot happen automatically, one would have to port it into the toolkit.

- Potential collaboration difficulties - as we, again, do not want to create dependencies on web services, collaboration would have to happen manually, with people sharing project data using other methods.

- Limited storage size - IndexedDB has limits to storage that tend to be smaller than RAM capacity.

3.3 Data model

In this section, we will explore the different types of data that simulus uses to model simulation cases. The data model of simulus loosely follows the CRISMA framework data model. There are many other types of data used than listed here, but we consider this the core domain of simulus.

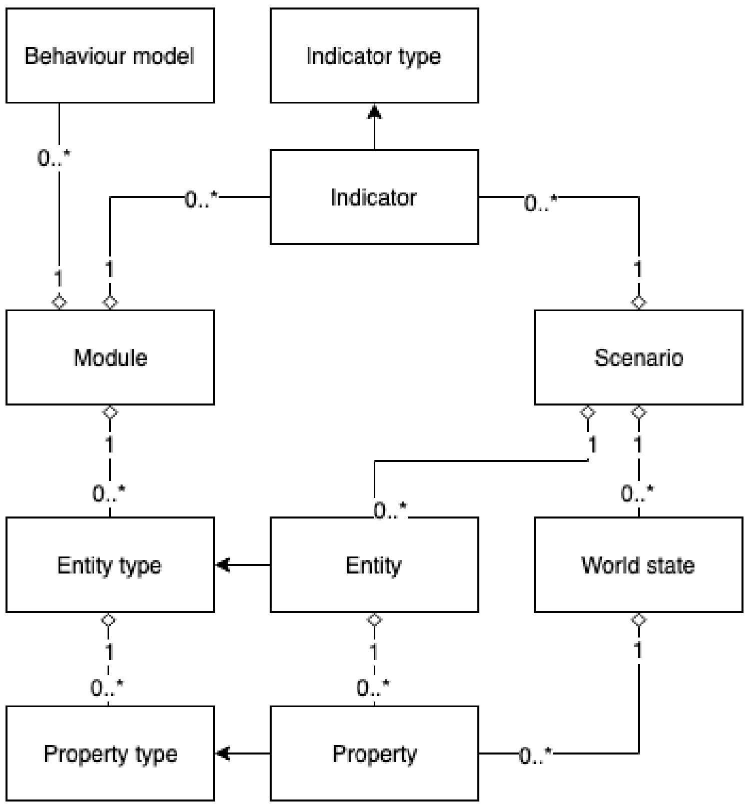


Figure 4: Overview of some of the data model

Scenario

The first concept to introduce is a scenario. A simulation always happens in the context of a scenario. Like in the CRISMA framework, a scenario contains an isolated instance of a simulation environment, independent from other scenarios. These scenarios contain all the data necessary to run, analyse and explore simulation cases. Scenarios are implemented as plain Javascript objects, and persisted in the client-side database, IndexedDB.

World state

A world state, responding to the same item in CRISMA literature, represents the entire state of the world and simulation at a single point in time. It is tied to a specific scenario. When looking at a scenario, it consists of a number of world states. A world state can be attached to a previous world state, and they can branch from each other, forming a tree. This is so that one can try going different paths during a scenario, to compare how they turn out, helping a user ”play” with the system. World states are immutable, if you want to change the state of the world by modifying it manually or running a simulation, you must create a new world state, usually based off of a previous world state, by means of a transition. World states are implemented as Javascript objects that store id-s of parent world states, constituting a tree. World states are also persisted in IndexedDB.

Transition

A transition represents the movement from one world state to another. It is used as an intermediary piece of data to construct a new world state from a previous one and a set of modifications to the previous one. Transitions are created by a user through the UI, by branching off of a previous world state and then editing it through manual editing and simulation. Unlike in CRISMA, transitions aren't persisted, they're only used as a temporary data structure to construct a new world state.

Entity type

An entity type corresponds to an ”OOI type” in CRISMA literature. An entity type defines what properties entities of this type have, and how they behave. In object-oriented programming, this responds to a ”class” of object. Much like OOI types, and classes in OOP, these types can extend each other, inheriting the properties of the parent. Entity types are not stateful, but they are used to construct entities, whose properties are stateful. Entity types are not provided by the platform, but created by a user programmatically in a module. Entity types are simple javascript objects, but since they can extend each other, there are facilities to collapse an entity type hierarchy into a single usable entity type.

Entity

An entity corresponds to an instance of an entity type, to an OOI in CRISMA literature, and to an instance of a class in OOP. Entities have properties of various types, that define their state. While entities are attached to and remain in a scenario forever, meaning that they themselves do not constitute state, their properties change throughout a scenario, constituting state. An entity models a single agent or actor, for example a single patient in a mass-casualty incident simulation. Entities can be created and archived via the UI or programmatically. Entities are javascript objects that are persisted on IndexedDB, and their metadata (not properties) can be edited at any point during a scenario's lifecycle. Contrary to the CRISMA model, entities do have a special geometry attribute, instead geometries are implemented as properties.

Property

A property is a piece of stateful data attached to an entity. Entities have a number of properties, which constitute the state of the entity. Properties are meant to change in a scenario, and their values are attached to world state, thus enabling one to see the change in properties. The types of an entity's properties are defined by that entity's type and its parent types. Properties can be of various types, for example, strings, numbers or geometries. Properties are javascript objects, they are created and changed when one is building a transition from one world state to another. As they constitute a critical part of the world state, they are immutable once stored in IndexedDB.

Indicator

An indicator is something that is able to translate a raw world state into a useful piece of visualised data. It is essentially function f with some added metadata, such that when fed a world state w, produces an indicator vector iv $f(w) \to iv$, where the indicator vector is a piece of ”visualisable” data, such as a single value, a grouping of labels and values we can show in different charts, or a ”custom” value, where the indicator will provide a UI component that can render this value. Much like entity types and behaviour models, indicators are not provided by simulus, but created by the user in a module.

Module

As previously discussed, we need some way for users to add their own behavioral code and models into simulus. This is where a module comes in. A module is a grouping of indicators, entity types and behaviour models that a user of simulus provides to their simulus instance. When creating a scenario, a user can select one or many modules from modules the user has loaded into their instance of simulus. These modules provide the custom functionality that a simulation needs. When running a simulation on one of the world states in said scenario, the user can select a behavioural model to run. The module abstraction is made so that simulation can be shared and integrated between each other by defining a common interface. When a scenario is created with modules attached, if that scenario is used in a context where those modules are not available, it will detect this and prompt the user to add these modules. A module is a simple javascript object that a user of simulus defines with all the indicators, entity types, behaviour models and metadata they want to include. It’s passed into the simulus framework to be used. The CRISMA data model doesn’t include modules, as its approach for custom behaviour is instead built on service composition.

Behaviour model

A behaviour model is another function with metadata, that, when selected by the user, can compute a new world state out of a previous one. Behaviour models are provided by the user in a module, and they constitute the actual algorithms that simulations use. To implement a behaviour model, one has to first decide what parameters (besides world state) their behaviour model needs. To allow users to enter these parameters when using the model, the model can provide a ”parameter schema” built out of JSONSchema, which is a standard format for defining data types [14]. Out of this schema, simulus will generate and render a form that will provide the user’s simulation model with the needed parameters. Finally, to make the model work, the user has to implement a function in the behaviour model that takes the world state (and associated metadata), and provided parameters (from the form we created from the scema), and returns a result asynchronously. The user can choose whether this function will be run synchronously or in a web worker, as there are some caveats in our implementation of web worker usage,for example, they can’t use variables out of the immediate scope of the function, as we have to serialize the function when moving it to the worker.

3.4 Implementation

Simulus applications are browser-only, and require no integration with web services or their API-s. A simulus application, when built, will be a set of static files that one can serve with any webserver. Simulus itself is built on a variety of client-side technologies and open-source libraries that affect its usage and development. We will list the more significant ones and their usage in this section.

Core technologies

Simulus is built using Typescript, which is a superset language of Javascript, meaning it supports the entire syntax set of the ECMAScript standard, but adds its own functionality on top [15]. Typescript provides static typing to an otherwise dynamic language. The reason for using it is twofold: first, it aids development, as it lets one use their code editor's autocomplete features in a more thorough way. Secondly, it aids in integrating simulus into your application, as someone consuming the simulus library can use a better-documented interface out of the box. To use typescript in the browser, it is transpiled into regular Javascript, using the typescript transpiler.

Simulus relies on a variety of modern browser features, which renders it unsuitable for older browsers, without including polyfills [16]. Importantly, simulus is meant to persist relatively large amounts of data in the browser. For this, simulus uses IndexedDB, which allows us to store and retrieve large amounts of data quickly, while allowing for different indices for our data models [17]. In addition, it provides transaction functionality. IndexedDB storage limits depend on the browser and their version and IndexedDB might not even be persistent in some specific cases. In case of firefox for example, the storage limit is based on the size of your hard drive, along with different ratios and constraints applied to it. However, a persistent thread between all the browsers is that the storage limit is not in hard-drive levels of capacity, but instead fits into RAM. This sets us an automatic upper limit on the amount of data we can work with, at least until newer solutions get standardised with larger capacity, like the File and Directories API proposal [18].

In order to run computation asynchronously and in parallel, simulus supports the usage of web workers for behavioural models [19]. As javascript is single-threaded, running heavy computation on the main thread would block the entire ui from interaction, which is a bad experience for users. Web workers help us avoid this issue, by moving computational work to a separate thread. Normally, web workers require separate files and special setup, but that is not in line with simulus' plug-and-play nature. Our implementation for this instead relies on serializing javascript, which means that when one is using the web worker mode of behavioural models, they have to pay special care that they don't violate its constraints. As this is a caveat, we also provide the possibility of running models synchronously.

Simulus has a need to store globally-unique identifiers, to not have conflicts when exporting or importing data. For this, we use Universally Unique Identifiers, otherwise know as UUID-s, which are in wide use in distributed and multiagent systems [20]. We use the version 4 (random) variant. For this, we're using an MIT-licensed javascript implementation, compliant to RFC 4122 [21].

User interface

A substantial part of the simulus system is the user interface (UI) that it provides to simulation managers. This includes utilities for managing scenarios, entities, world state, modules, running simulations and visual analysis. For building the interaction of the UI and our data model, we have used React, which is a UI library built by Facebook, that follows the functional programming paradigm [22]. It is licensed using the MIT license, which is a very permissive license [23].

For the actual visuals of the UI, in order to not spend much time on this, we have opted to use a component library called semantic ui, and more specifically, the official React adapter for it, which is also MIT-licensed. In simulus, we use charts to allow the user to visually analyse indicator vectors that their indicators provide. For these charts, we have integrated react-vis, which is an MIT-licensed charting library built by Uber, specifically with a focus on composable charts in React [24]. In addition, we have used d3, which is a ubiquitous tool for building visualisations on the web [25]. It is licensed using the BSD 3-clause license.

For convenient browser navigation, simulus uses an MIT-licensed abstraction on the browser history api, called react-router. This helps users navigate around the application. Its usage is also configurable by the user, as its default settings might not work in all environments.

Code structure

Simulus makes extensive use of the ECMAScript module specification, allowing us to split functionality into relevant files and directories and then having our build system add them together, while maintaining local scopes, without sacrificing loading speed. The code is split by domain, where things related to a single domain like world states or entities live together. A small simulus application (without much module code) takes about 600kb of space after bundling, minification and gzipping, which are best practices for web assets [26].

Simulus is open source, licensed under MIT, and available at the source code repository at https://github.com/tankenstein/simulus.

Items left out of scope

Some simple but useful facilities were out of scope for this thesis, but would be useful to include in the future. Things that have a good chance of being added to simulus include:

- More indicator types - currently, the variety of indicators is quite limited. Custom indicators help with this, but more indicators types would make better analysis more accessible.

- Scenario import and export - the simulus data model is meant to be easily serializable for this purpose. It would be very useful if a user could send their scenario in some serialized format to another user, who could import it into their instance.

- Parallelised indicators - we could leverage the same web worker system for indicators that we used for behaviour models.

4 Using simulus

In this chapter we will give an overview of how a simulation manager should actually use simulus to build their crisis simulation application.

4.1 Distribution

Simulus is distributed as a Javascript library. This means that in order to build an application using simulus, one has to acquire the built artifacts of simulus and use them in their code. There are two artifacts one needs to use simulus, the built javascript file and built css file. Both are minified to lower loading times. All dependencies of simulus are included in the built arifacts. There are more methods to acquire the built artifacts than listed here, but two main ones are highlighted.

Via node package manager

The node package manager or NPM is a ubiquitous package manager used in Javascript development [27]. It started as an unofficial package manager for the node.js Javascript runtime ecosystem, but eventually people started using it for browser applications as well.

Simulus is distributed on the NPM registry as the ”simulus” package. In order to install the latest version via npm, one has to first install a stable version of node.js and npm, and confirm that their environment is working as intended. Then the user has to navigate to their local npm project they are developing, and should run the following in a bash shell:

```
npm install simulus
```

In order to use simulus in this format, one usually has to include a build system in their project capable of bundling npm dependencies into their javascript. We suggest webpack, which is a common MIT-licensed built system. Once a build system is installed and working, one can get a hold of simulus using the ECMAScript module syntax, like so:

```
import simulus from 'simulus';
```

In addition, simulus has some styling attached to it that a user should include. When using a build system such as webpack, they can simply add the following to their scripts:

```
import 'simulus/dist/main.css';
```

Via a content delivery network

Unpkg is a content delivery network for files hosted on the npm registry. We can leverage it to include simulus in our website without having to familiarise ourselves with the complexities of the modern javascript toolchain. First, just include a html script tag with the following src property: https://unpkg.com/simulus/dist/main.js, like so:

```
<script
  type="text/javascript"
  src="https://unpkg.com/simulus/dist/main.js">
</script>
```

Then, also include the styling of simulus by adding a link element in the head section of your html document, with the following href property, like so:

```
<link
  rel="stylesheet"
  href="https://unpkg.com/simulus/dist/main.css">
```

Either way that one includes simulus, they will then want to proceed to using the main simulus exports as detailed in the next section.

4.2 Building on top of simulus

Once simulus has been included in a project, work has to be done to build a simulation application on top of it. To just have simulus up and running, one just has to run the simulus function with a html element they intend to render the application into and an object defining options. For example, if one wanted to render into the document body, they could do the following:

```
const options = {};

simulus(document.body, options);
```

Configuration of simulus happens through the options object. It defines only two keys, those being `routerType` and `modules`. In this section we will give an overview, but not completely detailed information about how to build on simulus, instead, interested parties should go to the documentation for this.

Up to date documentation is available at the source code repository, located at https://github.com/tankenstein/simulus

4.2.1 Routing

Simulus is built as a single-page application, and it uses browser history to move to different pages within it. This makes for convenient usage, as you can still use back and forward controls in your browser. However, there are some caveats with different routing styles. So, we provide three different router types you can use, depending on your infrastructure. To use this setting, include it in the options object, like so:

```
const options = {
  routerType: 'hash',
};

simulus(document.body, options);
```

The first one is `'none'`. This will always work, does not matter what your infrastructure is. This is also the default value. This will keep the routing state in memory only, meaning you cannot use url navigation at all.

The second one is `'hash'`, using a concept colloquially called ”hash” routing. This uses full URL-s, but adds them behind a # symbol, which means that our routing should not require changes from your webserver, as only the browser sees the hash. With this setting, you can have URLs like: `https://example.com/#/scenarios/`

The final one is `'full'`, which uses the full HTML5 History API. This requires some extra server configuration, as the client-side application would be responsible for controlling url paths. For this, the content root of your application should always serve the simulus application, regardless of what route you go to the server with. With this setting, you will have URLs like: `https://example.com/scenarios/`

4.2.2 Modules

Like discussed in the data model section of simulus architecture and technology, the core abstraction used to build custom functionality into your simulus instance is called a module. A module is a javascript object, being a collection of metadata, indicators, entity types and behaviour models, built by a user, that simulus can use to provide simulation functionality.

In order to do anything useful with simulus, one has to either acquire and include a module, or built their own. The idea of modules is that by defining a common interface with metadata that simulation developers can use to integrate their models, developers can share and extend these pieces of functionality. Thus, a full simulus application consists of the simulus framework and included modules. We encourage potential developers of simulus to share their modules, for example on NPM, so that others can use and build on top of their work. Modules can even use functionality from other modules.

This is an example module:

```
const ExampleModule = {
  id: 'ee.ttu.simulus.example.v1',
  name: 'Example module',
  description: 'Not useful module used to show how to build a module',
};
```

To use it, simply include it in your simulus options like so:

```
const options = {
  modules: [ExampleModule],
};

simulus(document.body, options);
```

Then, you can include this module in a scenario when you create one via the UI. However, this module isn't useful, it only defines the metadata of the module. Name and description are purely used for descriptive purposes, but the id is important, as it is used to see which modules scenarios are dependant on. It should be globally unique, so one can use uuid-s for this, or follow the java package convention. We also recommend appending the version to the id, if some versions of your module are not backwards compatible. Most objects with metadata in simulus require globally unique id-s.

Entity types

The ”meat” of a module are the entity types, the behaviour models and the indicators it exposes. All of these are defined as objects. This is an example entity type of a simple fire truck model, with properties denoting its location and remaining water amount:

```
const FireTruck = {
  id: 'ee.ttu.simulus.example.v1.truck',
  name: 'Fire truck',
  properties: [
    {
      id: 'location',
      name: 'Location',
      dataType: 'geometry',
    },
    {
      id: 'waterRemaining',
      name: 'Water remaining',
      dataType: 'number',
    },
  ],
};
```

Figure 5: An example entity type

There are more ways to configure entity types, like using the built-in inheritance support via the base type id parameter, but these are available in the documentation.

Indicators

Indicators and behaviour models are special, in it that they expose actual javascript functions in addition to metadata. For an indicator, you first select what type of indicator it is. This denotes what kind of value will be rendered, whether this is just some value, or a graph of some kind. Once you know the type, you also have to make your indicator vector data conform to that type’s standard. This is an example of a simple indicator that, when calculated, will show a bar chart with fire trucks aggregated by their water level:

```
const FireTruckChart = {
  id: 'ee.ttu.simulus.example.v1.truckWaterLeft',
  name: 'Fire trucks by water left',
  type: 'bar',
  calculate: state => {
    const counts = entities
      .filter(({ type: { id } }) => id === 'ee.ttu.simulus.example.v1
          .truck')
      .reduce((aggregate, truck) => {
          const { waterRemaining } = truck.properties;
          if (!waterRemaining) aggregate.empty++;
          else if (waterRemaining <= 25) aggregate['<=25']++;
          else if (waterRemaining <= 50) aggregate['50']++;
          else if (waterRemaining <= 75) aggregate['75']++;
          else aggregate.full++;
          return aggregate;
        },
        {
          empty: 0,
          '<= 25': 0,
          '<= 50': 0,
          '<= 75': 0,
          full: 0,
        },
      );

    return Object.keys(counts).map(key => ({
      x: key,
      y: counts[key],
    }));
  },
};
```

Figure 6: An example bar indicator

Indicators have different types of values they must return, depending on their type. In the above example, a bar indicator (one which will render a bar chart) will have to return an array of objects, where the objects have an x value and a y value. The system will use this function to compute new indicator vectors and render them when necessary in the analysis tab.

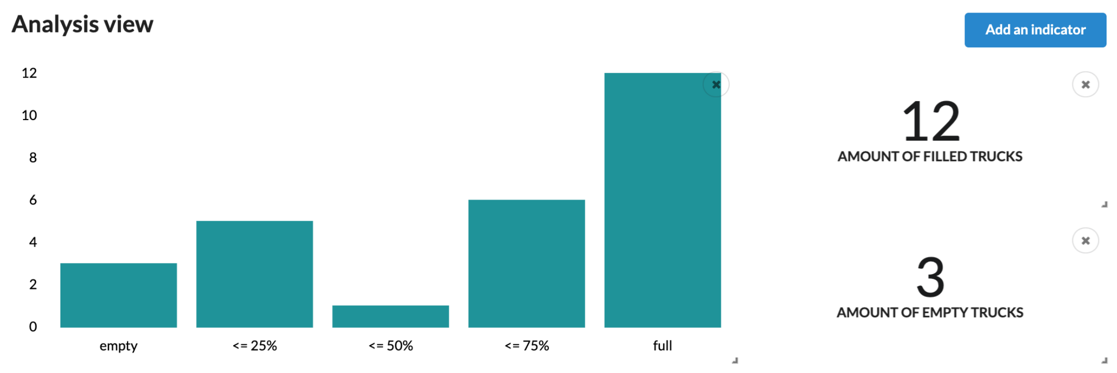


Figure 7: Simulus analysis including the previous example indicator

Behaviour models

Finally, to run simulation models, one has to implement them as behaviour models in their module. First, it's worth thinking about what data you'll need for this simulation. If it's not all contained in world state, it's worth defining some simulation parameters using the parameter schema option. This will generate a form for the user to fill when they want to run a simulation, and then pass the result to the model along with the state. Finally, when you have all the data you need, you need to implement the actual simulation function. This function takes in world state and simulation parameters and passes a new state to a function, effectively performing a state transition.

```
const WaterDecreaseModel = {
  id: `ee.ttu.simulus.example.v1.model.waterDecrease`,
  name: 'Simulate fire trucks randomly getting less full',
  parameterSchema: {
    type: 'object',
    properties: {
      steps: {
        title: 'Steps to run simulation for',
        type: 'number',
      },
    },
  },
  runInWorker: (state, params, callback) => {
    for (let index = 0; index < params.steps; index++) {
      state.entities
        .filter(entity => entity.type.id === 'ee.ttu.simulus.example.
          v1.truck')
        .forEach(entity => {
          if (Math.random() <= 0.2) {
            entity.properties.waterLeft--;
          }
        })
    }
    callback(state);
  },
};
```

Figure 8: An example behaviour model

If you'd like to make your model run asynchronously, you should use the `runInWorker` option instead of `run` when implementing the function. However, because of the way that we construct a worker out of the model, you cannot use variables outside of that function's scope. This is unfortunate, but we let you also use the synchronous `run` if this is an issue.

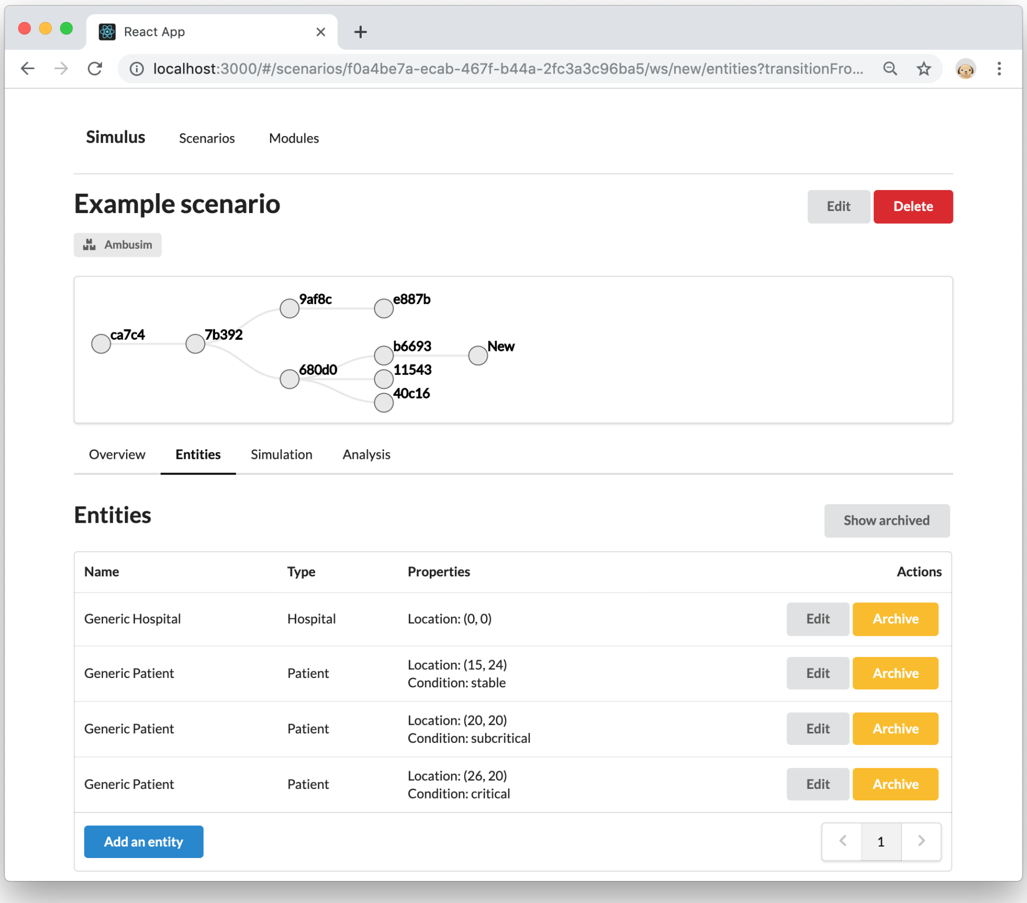
4.2.3 Using an application

A simulus application has three top-level views, and then many more subviews. When a user first enters a simulus application, they land on a ”landing” view. Although this view is out of scope for this thesis, it will be used to show documentation and information pertaining to the specific simulus instance. From here, a user can go to the modules or scenarios views. The modules view shows the user the modules they have currently loaded, and some of their properties. However, the most important view here is the scenario view. Here, a user can create and select scenarios to work with. In the future, this is where scenarios will also be imported and exported from.

Most of the time a user will be either creating or going to an existing scenario. This opens the main view, which is called the scenario view. Here, a user is first shown a tree of the current world states, which also acts as navigation between world states. Below this, the user can access all functionality pertaining to the current world state through four tabs:

- Overview - this lets the user create and delete new world states and transitions.
- Entities - in this view, a user can see and manipulate all the entities who are involved in the scenario, as well as their properties as they are in the current world state.
- Simulation - here, a user can select a behaviour model they want to run on the current world state, and will also fill the simulation parameters for these models. When they run an asynchronous simulation, they can monitor the progress here.
- Analysis - this tab is meant to analyse the current world state using indicators. A user can create panels that show indicators defined in modules here, and manipulate them with accessible facilities, like moving them around by dragging and dropping, or resizing.

A more detailed example of building a simulus module is available in the simulus source code repository, accessible at https://github.com/tankenstein/simulus.



Example scenario

Entities

Name	Type	Properties	Actions
Generic Hospital	Hospital	Location: (0, 0)	Edit Archive
Generic Patient	Patient	Location: (15, 24) Condition: stable	Edit Archive
Generic Patient	Patient	Location: (20, 20) Condition: subcritical	Edit Archive
Generic Patient	Patient	Location: (26, 20) Condition: critical	Edit Archive

Figure 9: Main simulus scenario view with the entity tab selected

5 Reference application

In order to validate the basic design of our framework, simulus, we have built a small reference application using it. The scenario we are simulating is a mass-casualty event simulation, with the goal of exploring how ambulances react to such an event.

There has been substantial research into ambulance simulation. In our case, we naively want to run a crisis simulation with ambulances, that is to say we will ignore all other happenings in the world, and assume that all ambulances are needed to handle the crisis situation. In addition, simulating dispatch is outside of the scope of this example module, so we assume all ambulances have perfect knowledge of the world.

All of the functionality described is built within one simulus module, called ambusim. In this module, we provide entity types to model patients, ambulances and hospitals. All the entity types described have a location property type. Patients have additional property types pertaining to their current medical condition.

We provide two behavioural models. One will simply create a mass-casualty incident, by providing the model with various properties used to describe the accident, such as its location, the area over which the victims are scattered, the amount of victims, the magnitude of the incident and so on. This model will create patients randomly with appropriate medical conditions in the accident area. For the ambulance model, we utilise an ambulance service process model inspired by the model used in ”BartSim: A tool for Analysing and Improving Ambulance Performance in Auckland, New Zealand”, but simplified [28]. When an accident occours, all ambulances start moving towards it. They select the closest patient who is currently in critical condition. Once arrived, they will stabilise the patient, and move on towards the next one. At the same time, patients who are in critical condition are slowly deteriorating.

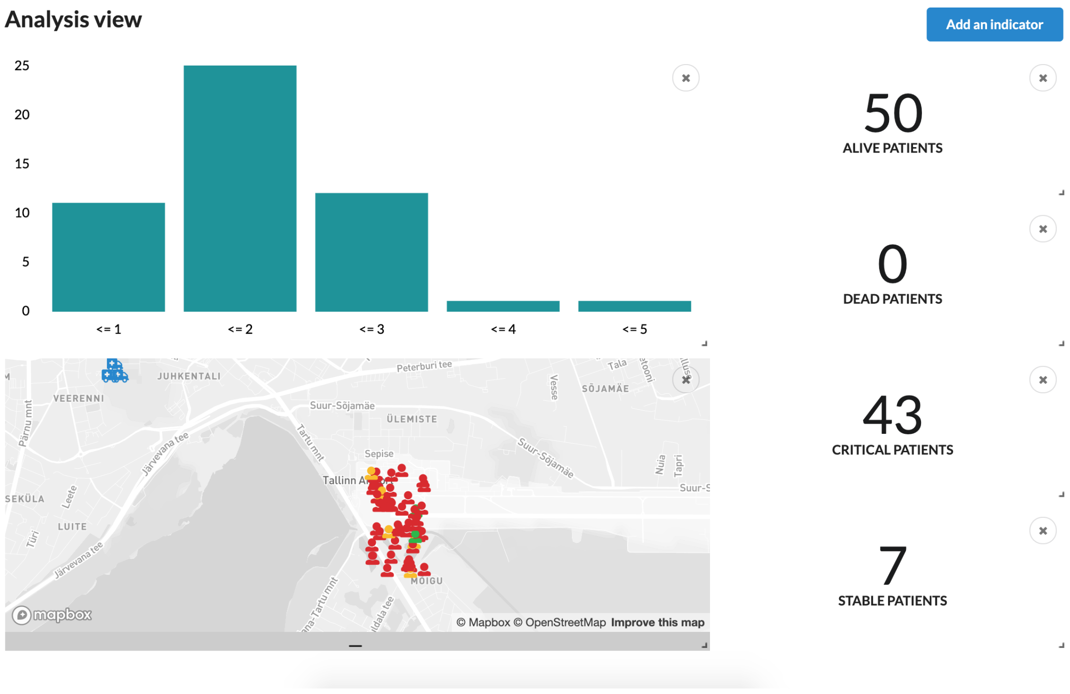


Figure 10: Start of a simulated mass-casualty incident

We provide some simple indicators to view the current situation in the world, along with a map indicator built with the Mapbox webgl environment [29], using OpenStreetMap [30] data.

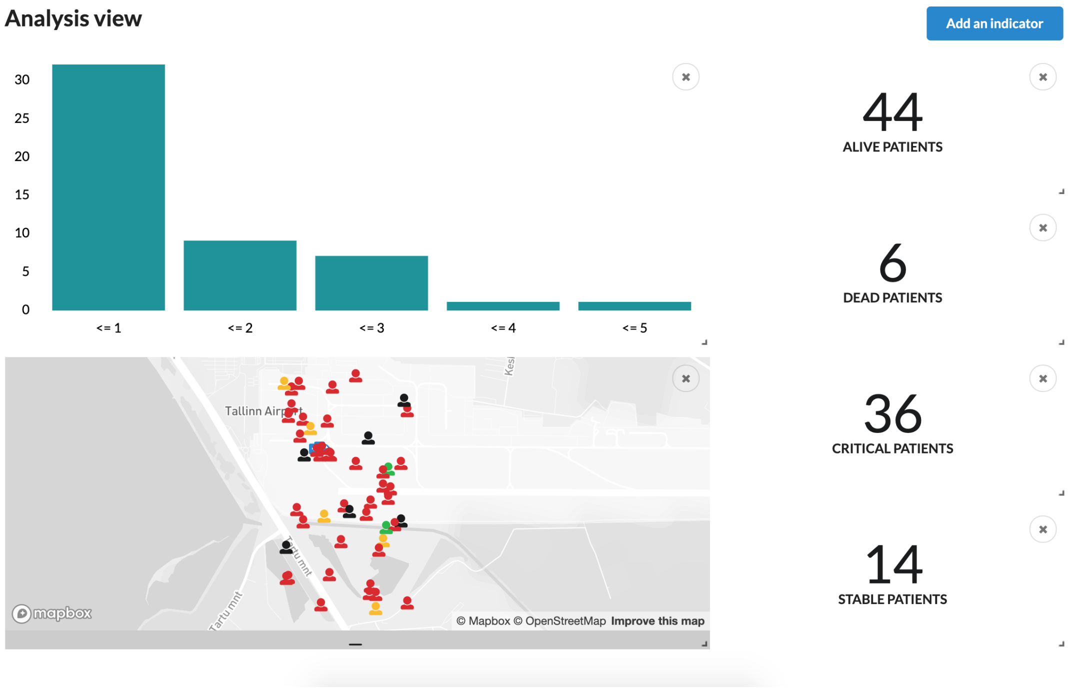


Figure 11: The same incident after 15 minutes passes in the simulation

From the above example, you can see how as the simulation progresses by 15 minutes from the previous situation, some patients have deteriorated, some have died and some have been stabilised by the ambulances that have arrived. Overall, the conditions have gotten worse. In this case, the amount of ambulances that have arrived during this timeframe are not enough, and stabilising all the critical patients is overwhelming the three ambulances currently on the scene.

A demo of this application, along with a small overview is available at http://tankenstein.github.io/simulus. The source code is available as an example in the main repository, https://github.com/tankenstein/simulus.

6 Summary

The field of crisis simulation is an ever more critical aid to planning and training for contingencies. We do have some very comprehensive toolkits at our disposal. However, due to the complexity of the field, these toolkits are not easily accessible and have a high barrier of entry: thus the same can be said about creating practical crisis simulation as a whole. In addition, most of the current research is geared towards applications useful for either serious research groups or large organisations. However, we believe there is value in making this accessible both for students and small and medium organisations.

In order to alleviate the situation, we have designed and built a crisis simulation framework SIMULUS with a focus on accessibility, not flexibility. We have based the design of SIMULUS on the CRISMA framework, adapting it for accessibility while limiting the scope. As a part of this focus on accessibility, we have built the framework entirely on browser technologies, not relying on separate databases, servers or web services. Technologically our system relies on several new browser API-s to mitigate some of the shortcomings of the browser platform. We have built a simple integration system into our framework to help users model their domain and to build simulations for it. In order to validate our framework, we have built a simple crisis simulation application which simulates ambulance response to a mass-casualty incident.

As future work it would be useful to validate SIMULUS with a real pilot application to see its shortcomings in practice, and then adapt SIMULUS to overcome those shortcomings.

References

[1] S. Fortmann-Roe, “Insight maker: A general-purpose tool for web-based modeling & simulation,” *Simulation Modelling Practice and Theory*, vol. 47, pp. 28 – 45, 2014.

[2] D. Havlik, P. Dihé, S. Schlobinski, A.-M. Heikkilä, M. Polese, K. Taveter, and O. Venho-Ahonen, *Modelling crisis management for improved action and preparedness*. 07 2015.

[3] C. M. Macal and M. J. North, “Tutorial on agent-based modeling and simulation,” in *Proceedings of the Winter Simulation Conference, 2005.*, pp. 14 pp.–, 2005.

[4] EU Publications Office, “Modelling crisis management for improved action and preparedness.” Available https://cordis.europa.eu/project/rcn/102347/factsheet/en. [Accessed: 2019-05-01].

[5] P. Dihé, M. Scholl, S. Schlobinski, T. Hell, S. Frysinger, P. Kutschera, M. Warum, D. Havlik, A. DeGroof, Y. Vandeloise, O. Deri, K. Rannat, J. Yliaho, A. Kosonen, M. Sommer, and W. Engelbach, “CRISMA ICMS Architecture Document V2,” 02 2014.

[6] M. Kleiboer, “Simulation methodology for crisis management support,” *Journal of Contingencies and Crisis Management*, vol. 5, no. 4, pp. 198–206, 1997.

[7] L. Sterling and K. Taveter, *The art of agent-oriented modeling*. MIT Press, 2009.

[8] J. Dugdale, N. Bellamine-Ben Saoud, B. Pavard, and N. Pallamin, *Simulation and emergency management*, vol. 10. Chapter, 2010.

[9] S. Abar, G. K. Theodoropoulos, P. Lemarinier, and G. M. O’Hare, “Agent based modelling and simulation tools: A review of the state-of-art software,” *Computer Science Review*, vol. 24, pp. 13 – 33, 2017.

[10] J. Byrne, C. Heavey, and P. Byrne, “A review of web-based simulation and supporting tools,” *Simulation Modelling Practice and Theory*, vol. 18, no. 3, pp. 253 – 276, 2010.

[11] EU Publications Office, “CRISMA Use cases.” Available http://www.crismaproject.eu/usecases.htm. [Accessed: 2019-05-02].

[12] P. Dihé, M. Scholl, P. Kutschera, A. DeGroof, O. Deri, A. Kosonen, and J. Sautter, “ICMS Framework V2,” 02 2015. [Accessed: 2019-04-28].

[13] I. Dasheysky and V. Balzano, “JWST: Maximizing Efficiency and Minimizing Ground Systems,” in *Proceedings of the 7th International Symposium on Reducing the Costs of Space Craft Ground Systems and Operations (RCSGSO)*, vol. 28, Citeseer, 2007.

[14] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, “Foundations of json schema,” in *Proceedings of the 25th International Conference on World Wide Web*, pp. 263–273, International World Wide Web Conferences Steering Committee, 2016.

[15] S. Fenton, “Typescript language features,” in *Pro TypeScript*, pp. 1–62, Springer, 2018.

[16] R. Sharp, “What is a polyfill.” Available https://remysharp.com/2010/10/08/what-is-a-polyfill, 2010. [Accessed: 2019-05-03].

[17] S. Kimak and J. Ellman, “The role of html5 indexeddb, the past, present and future,” in *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, pp. 379–383, IEEE, 2015.

[18] J. Bell, “File and directory entries api.” Available https://wicg.github.io/entries-api/, 11 2019. [Accessed: 2019-05-05].

[19] I. Green, *Web workers: Multithreaded programs in javascript.* ”O’Reilly Media, Inc.”, 2012.

[20] Z. Balkić, D. Šoštarić, and G. Horvat, “Geohash and uuid identifier for multi-agent systems,” in *KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*, pp. 290–298, Springer, 2012.

[21] P. Leach, M. Mealling, and R. Salz, "Rfc 4122: A universally unique identifier (uuid) urn namespace," *Proposed Standard, July*, 2005.

[22] C. Staff, "React: Facebook's functional turn on writing javascript," *Communications of the ACM*, vol. 59, no. 12, pp. 56–62, 2016.

[23] Y.-H. Lin, T.-M. Ko, T.-R. Chuang, and K.-J. Lin, "Open source licenses and the creative commons framework: License selection and comparison," *Journal of Information Science and Engineering*, vol. 22, no. 1, pp. 1–17, 2006.

[24] Uber inc, "React-vis: a composable charting library." Available https://uber.github.io/react-vis/. [Accessed: 2019-05-04].

[25] J. Harper and M. Agrawala, "Deconstructing and restyling d3 visualizations," in *Proceedings of the 27th annual ACM symposium on User interface software and technology*, pp. 253–262, ACM, 2014.

[26] S. Souders, "High performance web sites," *Queue*, vol. 6, no. 6, pp. 30–37, 2008.

[27] P. Teixeira, *Professional Node.js: Building Javascript based scalable software*. John Wiley & Sons, 2012.

[28] S. G. Henderson, A. J. Mason, *et al.*, "Bartsim: A tool for analysing and improving ambulance performance in auckland, new zealand," in *Proc. of the 35th Annual Conference of the Operational Research Society of New Zealand, Wellington, New Zealand*, pp. 57–64, 2000.

[29] O. Eriksson and E. Rydkvist, "An in-depth analysis of dynamically rendered vector-based maps with webgl using mapbox gl js," 2015. Dissertation.

[30] M. Haklay and P. Weber, "Openstreetmap: User-generated street maps," *IEEE Pervasive Computing*, vol. 7, pp. 12–18, Oct 2008.