Mikhail Bobretsov

DISPERSED CLOUD RENDERING

Bachelor's thesis

Information Technology

2020

South-Eastern Finland
University of Applied Sciences

South-Eastern Finland University of Applied Sciences

Author (authors)	Degree title	Time
Mikhail Bobretsov	Bachelor of Engineering	May 2020

Thesis title
Dispersed cloud rendering — 66 pages

Commissioned by

Observis Oy

Supervisor

Timo Hynninen

Abstract

The main goal of this bachelor's thesis was to improve the existing solution of rendering the three-dimensional dispersed clouds for a flagship product developed at the commissioner company Observis Oy. The current solution shows the 2D flat layers of a three-dimensional cloud, where every layer is switched by a button. The thesis was aimed to creating the Electron web application with the Mapbox map system and an extra layer of 3D clouds on top of it. In this case, the cloud will be shown as a whole 3D object with no need to switch layers and lose attention to perform unwanted actions. It should have interaction capabilities like observing the cloud state in a different point of time and automatic cloud animation.

The project's core development stack was Electron framework, React UI library and TypeScript programming language. The main visualization technologies were Mapbox for a map providing and deck.gl for rendering a three-dimensional point cloud layer. A special algorithm for processing multidimensional raw data was created. It prepared coming information to separated points of data with the exact geographical coordinate, material concentration value and calculated colour in multichannel mode. The interpolation technique was studied and implemented in practice. However, the Electron environment could not handle the interpolation workload and the whole codebase was migrated to a website environment with no rewriting code. A custom algorithm for interpolating cloud points was created, it successfully applied linear interpolation within three phases for three-dimensional clouds. The applied technique improved the visualization of clouds by generating more points between existing ones. It makes clouds to be denser and to look more realistic and appealing.

The resulting application delivered a new vision of the same data. Every dispersed cloud is represented as a set of thousands of coloured spheres all together forming the realistic cloud shape. Moreover, a unique user experience came true in observing cloud state for different time periods with manual hour slider and automatic animation.

Keywords

Dispersed clouds, deck.gl, Mapbox, React, TypeScript, WebGL, linear interpolation, SILAM, point cloud

CONTENTS

1 INTRODUCTION

Modern technologies are generating tons of complex data around the world. This data by itself may seem pointless, but the latest openings in data science show that it can hide very useful statistical information. It can explain uncertain events in the past or predict upcoming changes in the future. The main problems are how to appropriately process the data and then comprehensively show the result. Data handling is developing now actively, but the problem of showing the result is still opened. The difficulty here is that most solutions show plain tables with calculated outcome lacking interaction and the possibility to look on it from a different angle. Recently, big IT companies have started creating open-source software for visualization, which is capable of working with huge chunks of information, especially for web applications. One of the known cases is vis.gl open-sourced data visualization library set by Uber. Right now, the development of new solutions is opening a new horizon for data understanding.

Observis Oy's flagship product is the Situational Awareness System (SAS) software. It implements many innovative features, one of them is rendering dispersed clouds of different gases right on top of the map. It helps to detect the contaminated areas by showing the location of the cloud. Also, the cloud is divided into different coloured sectors depending on the concentration of the gas. It helps to better estimate the situation and make the right choice. However, the current solution for visualizing dispersed clouds is made in 2D where every layer is switched by a button. It cannot show the cloud as the whole object and force a user for unwanted extra actions. This thesis investigates the way to improve visualization of dispersed clouds by rendering them in true 3D. Algorithms for implementing visual part and data processing will be created.

The SAS product contains many modules and the main one is the map. This component provides a lot of essential information for a user and every new element on it should be clear to understand. The utilized map is web-based Mapbox system which supports 3D layers. My motivation is the possibility to work with the map system and to find a way to process and organize the existing data.

Another reason is to get experience with 3D data visualization toolset for web-based applications and to use gained knowledge in future work.

The main goal of this thesis is to improve the Observis Oy existing solution by rendering 3D clouds out of the same data that is used in the current solution. The second goal is to deliver better user experience with extra interactions, add the possibility to explore clouds at a different point of time or even animate it. Overall, web-based application with Mapbox map system and data visualization toolset will be developed. The purpose of the whole project is not to be a final solution but to be a next step in the evolution of the current approach and deliver a new better experience.

The theory part describes the software providing data for forming a dispersed cloud, the project environment with mentioning key frontend and backend libraries and frameworks, visualization tools and the technology that makes it possible and the main visualization concept which will be used for visualizing dispersed clouds.

The practical part gives an introduction of the project setup and the visualization tools configuration. Next, the data processing technique is explained in high details. After the implementing data processing algorithm, resulting visualization is shown. Lastly, cloud visualization goes beyond available data with the proposed way of generating more data.

2 THEORY PART

The theory part bases on the definition and description of the main technologies used in the project. At first, an introduction is given for the software providing data for the thesis project. What data it can calculate, which configuration files are needed and how the result is formatted? Next, the project environment is explained along with the programming language and backend and frontend setup. It introduces core frameworks and libraries on top of which the whole functionality is built. Further, tools for data visualization and technology what they based on are described. It gives showcases of the technology used and explains

how it is already changing ways of interacting with web apps. Finally, an approach for the dispersed cloud dataset visualization to achieve 3D experience is specified. A detailed description of the visualization technique and how it correlates with existing data of dispersed clouds is explained.

2.1 SILAM

The following section gives an introduction to software that produces the data which is used in this project. It is used for creating the clouds and it has all the essential information like coordinates and material concentration. This software will not be used in this project, but it is important to understand where the data comes from.

Data for dispersed cloud rendering is provided by SILAM. A dispersed cloud represents unevenly dispersed material in the air. Therefore, the cloud is shown not as one object, but a bunch of smaller clouds of the same material. SILAM (System for integrated modelling of atmospheric composition) is open-code software provided by Finnish Meteorological Institute with free use for research applications. It is a global mesoscale model which uses mathematical equations for determining atmospheric physics and dynamics. Based on this, SILAM is a dispersion model for computing atmospheric composition which, for example, can be used for calculating air quality, emergency alarm applications and inverted dispersion modelling software. (Finnish Meteorological Institute 2014a.)

SILAM documentation says that it can calculate “496 different nuclides, together with their radioactive decays and transformations; inert and chemically active size-specific aerosol; biological material (pollen grains); chemically active gases”. Also, it can estimate the measurement on-site using previously collected measurements. Figure 1 illustrates the forecast for SO2 gas over Europe. In this case, the system predicts the situation for three days in future. (Finnish Meteorological Institute 2014b.)

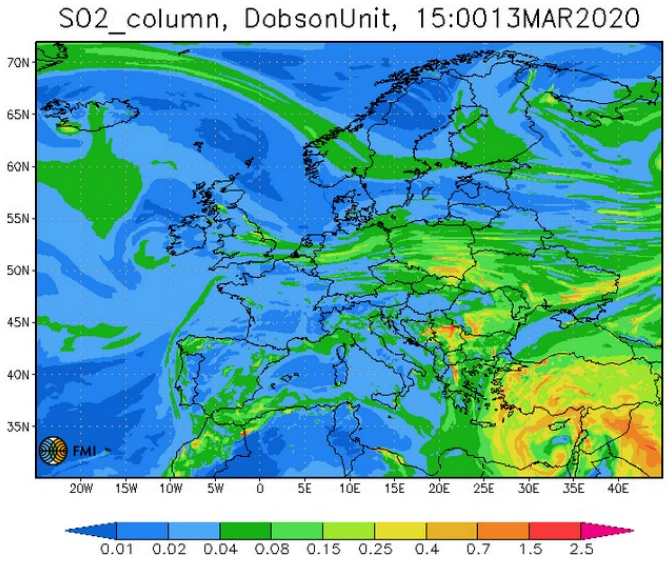


Figure 1. Forecast for SO2 concentration (Finnish Meteorological Institute 2014)

Single material or a mix of dispersed materials are defined as “cocktail” in SILAM. Additionally, it needs to contain some mandatory configuration files for a proper run such as the **control file** for run parameters, the **source term file** for emission sources and the **output configuration file** for output setup. The **standard cocktail file** defines the standard and user-created cocktails. Moreover, a configuration can contain nuclide, optical and chemical properties, and based on all that provided information the system is creating the simulated data. (Finnish Meteorological Institute 2020.)

After the successful run, the output file is represented in netCDF (Network Common Data Form) format, it is an open standard for creating, accessing and sharing multi-dimensional scientific data. This format contains a header which defines the layout of the data arrays and metadata with additional information in key/value form. (Unidata 2020.)

The SILAM output data is already processed to JSON format for this thesis. It gives better integration with web tools as it is a common format for exchanging the data between web applications. Moreover, the JSON format saves the data in a key/value form as the netCDF format. Therefore, not a single piece of important data will be lost.

2.2 Project environment

This section provides a description of base technologies for creating the project environment. Huge chunks of data need to be both easy to handle and visualize. The project environment is illustrated in Figure 2. The Electron environment is the base for everything. It has two main sections. The backend side will be responsible for the data processing logic. It will read the SILAM data from the JSON file, apply processing algorithms to it and send the prepared data to the frontend side. The frontend side, on the other hand, will take processed data and put it to the rendered graphical content in the application window. The communication between the frontend and the backend is organized by the Electron runtime, because it runs two separate processes, one for application internal logic, another one for UI rendering. They cannot communicate directly, but the Electron environment takes this responsibility.

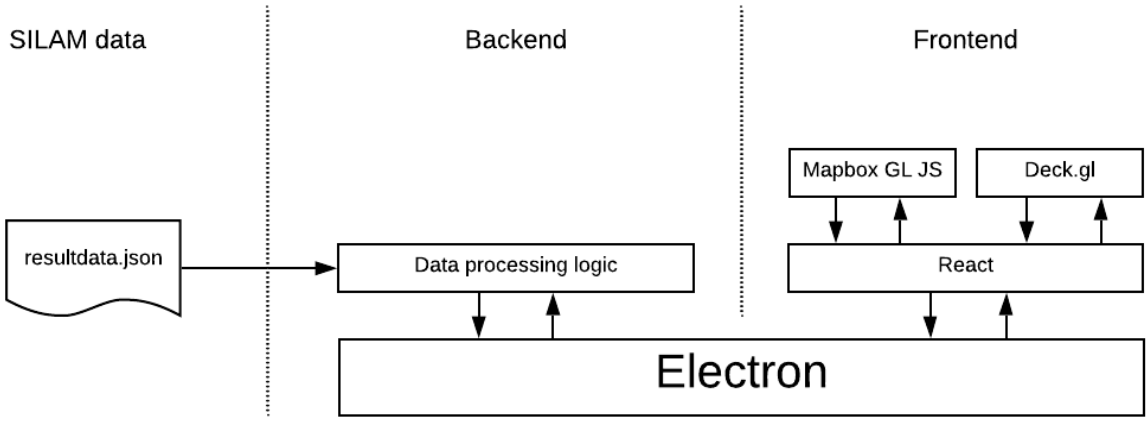


Figure 2. Project environment

Tools for the project must base on web technologies. It makes further integration to Observis' product easy and reduces development time with the well-known pipeline. Therefore, technologies for project accomplishment will be selected with high attention.

2.2.1 Electron

Electron is an open-source library created and developed by GitHub. Under the hood, it combines Chromium and Node.js to make building cross-platform apps possible in a single runtime with HTML, CSS and JavaScript. This stack of

technologies allows building the application for three main platforms: Windows, Mac and Linux. (GitHub 2020a.) In a few words, the Electron application opens a website shell which looks like a distinct program. It has access to the operating system's native API. All the content is rendered in the Chromium open-sourced browser, that is why the web technologies are used. Therefore, there is no need to be bothered about supporting other operating systems, the Electron framework makes it out of the box.

At first in 2013 Electron started as a dedicated framework for Atom code editor. It was popular for the very easy approach of creating extensions, as it does not require any prior knowledge besides some experience of web technologies. Later, Electron became so popular that many big IT companies created their applications based on it. For example, the text editor Visual Studio Code and messaging application Skype by Microsoft are created as separate programs, running similarly on different systems and utilizing web technologies for their logic.

Nevertheless, Electron has some drawbacks like application size and code security. Firstly, for a relatively small program, its dependencies can take a sensible amount of space. Secondly, as the app runs in a browser shell anyone can read the code. Nonetheless, these are just specific of Electron, since modern computers have a big amount of storage and source code is obfuscated for every program build.

Fortunately, the technology tandem in Electron suits the project needs appropriately. For example, Node.js runtime is working fully asynchronously and makes the user experience smooth even under high loads. Chromium, on the other hand, is the open-sourced browser from Google. It allows creating user interfaces with well-known web technologies which look the same on different operating systems. Moreover, Electron can separate the main process and rendering processes for every screen, so that they don't interfere with each other and use system resources effectively. Additionally, it supports native APIs for operating systems in managing folders and files. (GitHub 2020b.)

With Electron it is possible to use the well-known web development technology stack due to Chromium UI rendering engine. Moreover, it gives a boost in logic processing as the runtime is separated from the renderer. Therefore, web apps have more delay in data calculation and handling in comparison with Electron apps. However, a project based on Electron has a single main process managing all windows, interactions and operations inside the app. In case of heavy CPU task, it can be blocked and the app will be non-operational until the task is completed.

Overall, Electron intensive capabilities will help on every project step from data handling to its visualization. The main logic for data processing will be created in the Electron runtime.

2.2.2 React

As the solution for backend implementation was chosen in the previous section. In this section will be observed a technology for creating UI.

An interface for the project application can be built with a plain HTML, CSS and JavaScript stack. Nevertheless, almost all average user's web apps are created with some external frontend framework. They make a web app more reactive and smoother, use fewer system resources and advance web app development.

One of the popular and heavily used frontend frameworks is React. It started as an internal library for Facebook products and became open-source in 2013. On the release the Facebook development team propose some key benefits that React brings for UI development:

1. React does not use a template approach for building an interface. Every UI element is represented as a separate instance called "component", it is built with JavaScript. Also, they introduced an extension with the enhanced syntax for JavaScript called JSX.
2. By unifying markup with following view logic, React is making it easier to extend and maintain views.
3. Unified UI logic into JavaScript doesn't allow manual string concatenation and by this reduce the possibility for XSS (Cross-site scripting) attack. The

closed vulnerability enables the attacker to inject malefic script for later execution on client side.

Additionally, React implements a new approach in DOM manipulation, different from traditional JavaScript web applications. DOM (Document Oriented Model) represents a hierarchy of all interface elements with their content in web application. React uses so-called “virtual DOM”, it doesn’t refresh the whole DOM for single element update but inserts the visual changes to the existing DOM tree in the correct place. This process is so fast that explicit data binding is not needed. This approach helps to create modern web solution faster and easier with React. (Pete, Hunt, 2013.)

One of the main approaches of React is props and state use. Props (short for “properties”) characterized as data, hierarchically passed from a parent component to a child component. Props is a read-only data that can be modified only by the parent component. When props change, it triggers the child component to automatically re-render so that present data is automatically shown to the user. State, on the other hand, is the data that personal for every component and it can be modified. State change also triggers a component to re-render and state update may happen asynchronously. Moreover, state data from the parent component can be passed as props to a child component what makes data handling very flexible. (Facebook, 2020.)

According to Facebook, each React element has lifecycle methods for executing code in the exact time at triggered events, each of them can be overridden. The main lifecycle method is **render**. This method aims to display the passed data and return the result. The returned data type from **render** can be a React element or an array of React elements, string and numbers, or nothing at all. The React element lifecycle is divided into four phases that can happen multiple times or never at all, every phase contains its method for execution. Following Facebook React documentation, these phases are as follows:

1. **Mounting** methods called when a component is about to be created and put into the DOM
2. **Updating** methods called when the update is about to happen, and the object is being re-rendered

3. **Unmounting** method called when the component will be removed from the DOM
4. **Error Handling** methods executes when there is an error during component rendering

During component lifecycle it is not necessary to implement all the methods. However, there are some methods that very often implemented and mostly used for working with data:

- constructor() is called right before mounting the component. The good spot for the state initialization and data binding.
- componenDidMount() is invoked after the component is mounted. The right place for data loading and initializing network requests.
- componentDidUpdate() is called after updating occurs. It can be either props or state update. At initial render this method is not called. Good opportunity for conditional code execution as it allows comparing current state and props with previous ones.
- componentWillUnmount() is invoked before the component is unmounted and destroyed. It is the place for implementing any necessary clean up like stoping timers and cancelling active network requests.

Figure 3 the best illustrates when described above lifecycle methods are called.

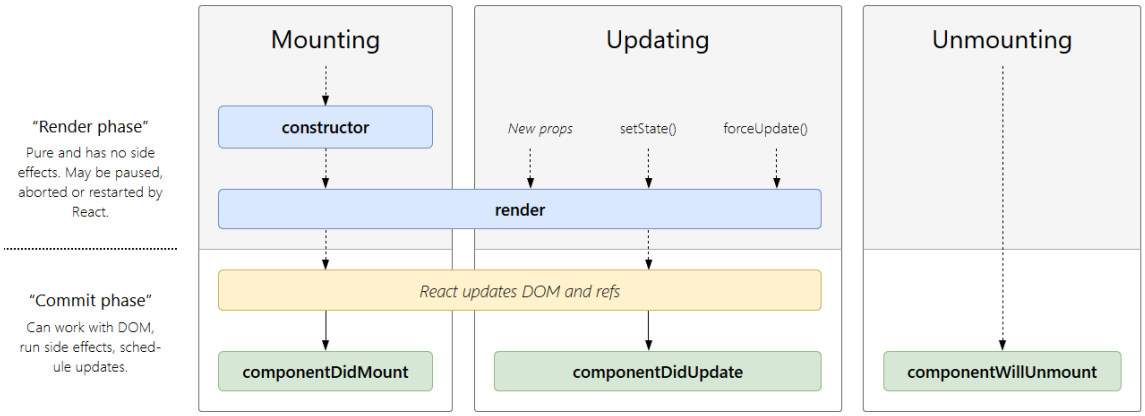


Figure 3. React component lifecycle diagram for version 16.4 (Wojciech Maj 2020)

Understanding of described above phases will help in organizing data communication logic between components. As a result, all these strategies for creating UI provided by Facebook in React library makes it easy to deliver good looking and fast web applications without any struggle.

2.2.3 TypeScript

Observis is always aiming to produce reliable software. It is impossible without creating a strong and understanding code base. Therefore, all the code instead of JavaScript is written with TypeScript.

TypeScript is an open-source programming language created by Microsoft. Syntactically it is a subset of JavaScript, which is adding static typing to the code. One of the characteristics of JavaScript is that it is dynamically typed. Dynamically typed languages are those where an interpreter assigns a type to a variable during a runtime when the value is set to the variable. Therefore, it adds more flexibility in handling one variable by assigning different types of data, but also it adds difficulty for software developers. It makes difficult to accurately evaluate variable data type at the development process. For example, some function receives a variable called “book”, immediately developer raises questions like: “Is it a name of the book or what?”, “Is this an object with arguments?”, “If it is an object, what are types for its arguments?”. Eventually, it leads to more errors in the code, more debugging and unnecessary comments. However, all these inaccuracies are eliminated with TypeScript. (Microsoft 2020a.)

The major difference between TypeScript and JavaScript is that this language adds static typing. For statically typed programming languages, the data type associated with the variable is known at compilation before the actual program runtime. In the result, code written with TypeScript tends to fewer human errors, it is easier to read and maintain, less unnecessary testing is required, and all of these is possible in the same JavaScript ecosystem. To achieve it, the TypeScript code is transcompiled to JavaScript. Transcompile operation means the process of translating source code from one programming language to another. (Microsoft 2020b.)

TypeScript compilation is handled by own compiler called **tsc**. It can compile files locally with the provided instructions or apply its default parameters. To unleash the whole compiler potential, it is a recommended approach to include

configuration file “tsconfig.json”. It defines compile options like paths to the source file and paths for output files, source code error checks, modules importing rules and many more. In the result, all TypeScript files with “.ts” extension are easily transcompiled to JavaScript files with “.js” extension. (Microsoft 2020c.)

Moreover, TypeScript supports libraries written in plain JavaScript. For making them operational the declaration files must be included. Declared variables and functions are reachable for the **tsc** compiler and act as the bridge between compiler and JavaScript library code. Unfortunately, major changes in the library should be checked and sometimes rewritten in declaration files, otherwise, it can lead to code malfunction or unpredicted results. (Microsoft 2020d.)

Use of TypeScript will ease the understanding of data structure from SILAM output. TypeScript will help in creating a more predictable and secured code base. Also, it can reduce the time for the possible code migration to Observis product.

As a result, the choice in favour of the TypeScript should be fully justified. With predictable and secure code behaviour it adds an extra abstraction level and sacrifices JavaScript code flexibility. Nevertheless, TypeScript helps in Observis’ intend of delivering reliable software and this choice is fully reasonable.

2.3 Visualization tools

This section describes two major components for visualizing the processed data and the technology which they are based on. Visualization needs to be represented as full 3D objects on the map. It should be interactable from the box and integrable to the project. Therefore, visualization must respond to the user’s actions like zooming or tilting the view. In this case, it shows the whole potential of 3D clouds, when the user can see it from another angle or zoom to the desired area of the cloud.

Both of them are React compatible JavaScript libraries and can easily work in a thesis project's environment. Moreover, for view rendering they are using the promising and game-changing WebGL technology. Alongside with new capabilities for web developers, its feature set is bringing a new experience for web users which was never unattainable before.

2.3.1 WebGL

WebGL (Web Graphics Library) is a JavaScript API for rendering resource-intense 3D and 2D graphics natively in the modern web browsers. It is possible by using OpenGL ES (OpenGL for Embedded Systems) in canvas element introduced in HTML5. Canvas supports drawing graphics, it can combine pictures, draw graphs or create different animations usually with JavaScript. (Mozilla Corporation, 2019.)

OpenGL ES is a subset of desktop OpenGL, a specially designed API for rendering advanced 2D and 3D graphics on embedded systems like phones, game consoles, tablet and car computers. It can be used on low-powered systems and supports hardware-accelerated rendering using GPU (Graphics Processing Unit). WebGL is available in two versions: WebGL 1.0 supports OpenGL ES 2.0 API and WebGL 2.0 supports OpenGL ES 3.0 API. OpenGL ES 3.0 is a noticeable update. For example, it adds new texture capabilities and can handle multiple rendering targets. (Khronos Group, 2020.)

Today WebGL is a worldwide standard supported by all major web browsers, and giant IT companies like Apple, Google, AMD, ARM, Epic Games, Intel, Nvidia and Qualcomm. They actively participate in the development and have voting right in Khronos Board (Khronos Group 2020). Therefore, the ease of use and openness of this technology makes it available for smaller companies and individuals for creating their products and tools based on WebGL.

WebGL is used in various software. One of the known appliances is Google Maps. It started using WebGL as an experiment in 2011 and already the pre-release version achieved true 3D buildings, smooth transitions in zoom levels and

StreetView, and now the whole map is a one canvas web element with rendered graphics. (Google 2020.) Other interesting cases are IKEA's planning tools. They allow creating kitchens from scratch in pure 3D with hundreds of furniture models made by IKEA, and all happens natively on the web. (IKEA 2020.) Such examples are just a quick peek into a new market of the web projects introduced by extensive capabilities of WebGL and community support. Showcases are varying from a pixeled game in VR built with a real game engine up to anatomy accurate human brain models with head and neck muscles which consists of more than three hundred structures illustrated in Figure 4. (Open Anatomy Project 2020.)

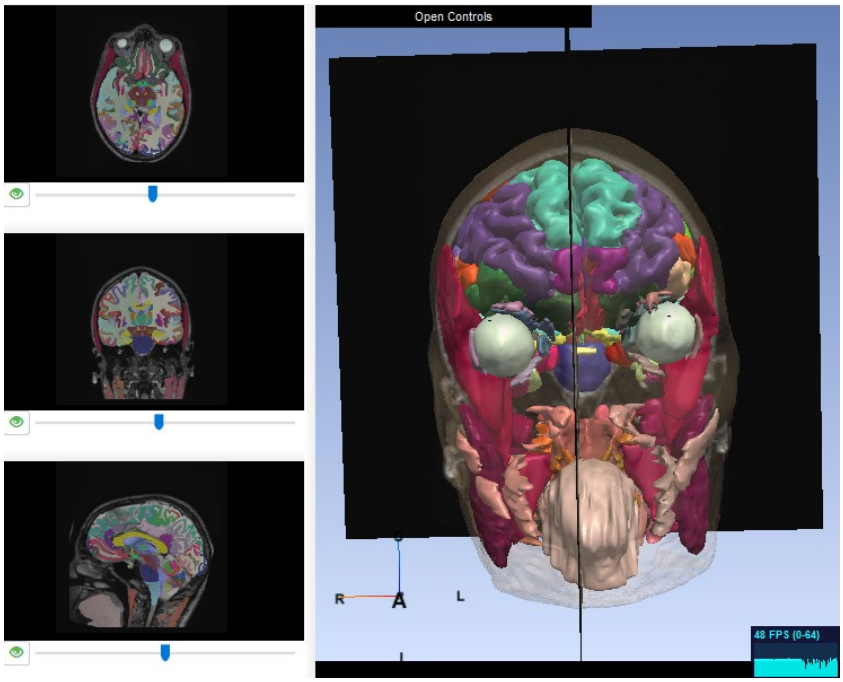


Figure 4. Brain MRI (Open Anatomy Project 2017)

For importing WebGL technology there is no need to work with its bare API. Based on this technology advanced libraries for 3D and 2D rendering are created, such as Three.js for object rendering and Luma for data visualization. They bring all WebGL capabilities in an accessible way for everyone without the need for any prior knowledge about textures, shaders and render engines.

2.3.2 Mapbox GL JS

Maps are heavily used in every Observis product, especially in the software this thesis is aimed to. In this software the Mapbox system is used. The map is

placed exactly in the centre when the user launches the app (see Figure 5). The map is a very important source of information for customers, it shows the updated and detailed overview of the current situation in the desired area.

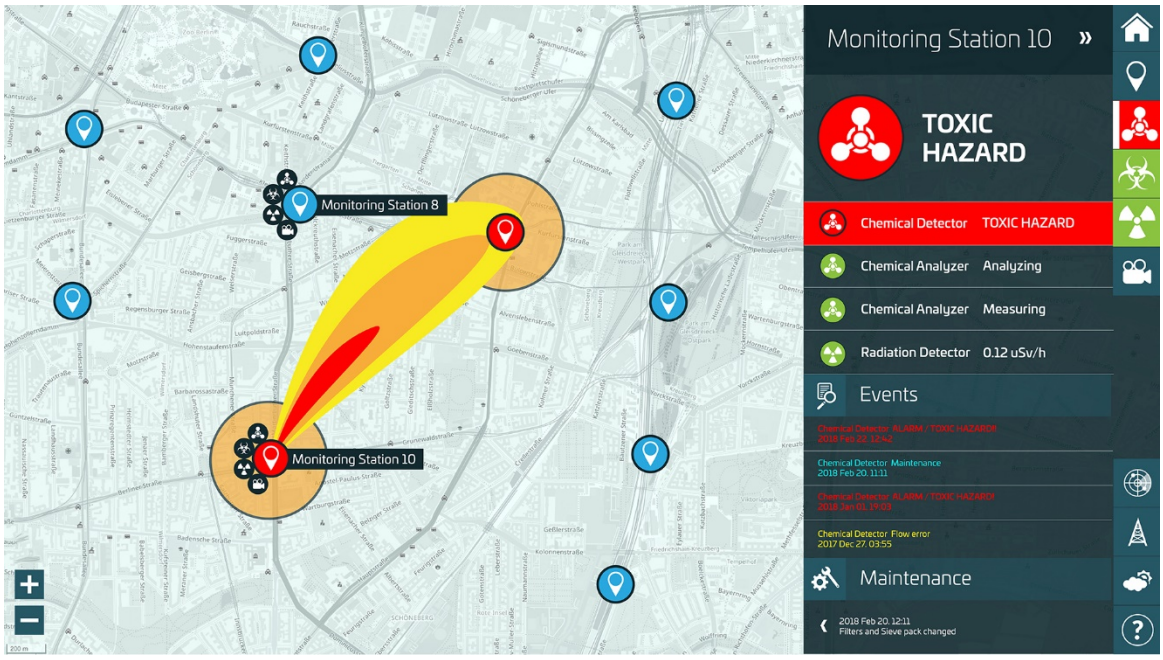


Figure 5. Observis SAS application

It should be not only detailed but clear for a user. For this software, a special colour palette is used that reflects the situation status for the area. With this approach, it is enough just for a quick glimpse on the map to estimate the situation based on the objects' colour. For a more detailed overview, a customer can zoom the map and see more custom made objects that look distinct and all together illustrate a better perspective on the situation. Summing all up the above, the map solution needs to be very flexible and to be built with modern technologies. One of the best solutions meeting the requirements is Mapbox GL JS.

Mapbox GL JS is a JavaScript library based on WebGL for rendering interactive maps. It is one product from the Mapbox GL ecosystem. The ecosystem includes solutions for Qt, Unity, iOS and Android. Alongside with powerful maps API, there is a Mapbox Studio application for creating fully custom maps. All these provided solutions are created by a private American company Mapbox. Dozens of customers are integrating Mapbox software into their applications and websites.

The most famous cases are Facebook, Snapchat, The New York Times and The Washington Post. (Mapbox 2020a.)

This map system widespread use is based not only on distribution for different operating systems or on SDK variety but also on the flexibility of working with different kinds of data. For any web application Mapbox GL JS can be imported, as it is based on WebGL technology and all of the map renderings are created on a single canvas web element. Moreover, the Mapbox development team made it possible to render custom or premade map layers for showing any geolocation data more clearly. They are rendered on the same canvas web element with the map. Additionally, due to WebGL integration, map layers can be rendered in full 3D with object perspective and simulated light. These features give full immersive experience for map users, which varies from finding new friends on the map in Snapchat or observing integrated map in a news article which is showing states vote distribution for a new American president. (Mapbox 2020b.)

Nevertheless, available premade layers lack a point cloud layer for displaying dispersed data. Fortunately, extensive Mapbox GL JS API allows enthusiasts and other companies to create their map layers. One of the companies which will be described below has developed a map layers library that fulfils the requirements for this thesis.

2.3.3 Deck.gl

Deck.gl is a high-performance WebGL framework for visualizing big data in 3D and 2D. It is one of the main frameworks from vis.gl ecosystem developed by American public company Uber. (Deck.gl 2020.)

Key emphasized features of deck.gl are: 1) resource efficient rendering of big datasets 2) event handlers for interacting with rendered objects 3) integration with major map providers 4) library of well-tested layers. Deck.gl can be imported into a project as a standalone JavaScript library or as a React component. (Uber, 2020a.)

Fortunately, right from the box, deck.gl has integration with Mapbox GL JS. It can add any available deck.gl layer as a custom layer to an existing map. Also, the deck.gl development team created a custom wrapper for a Mapbox map in a React environment. In this case, it creates a second transparent canvas element on top of the map, which renders all the layers in a more efficient way independently of the map. (Uber 2020b.)

Deck.gl has an extensive customizable layers library. A layer is represented as additional information floated on top of the map, it can be 2D or 3D objects, lines or text. Multiple layers can be rendered on the same map element as each has a unique identifier. Moreover, deck.gl has integration with Mapbox, which is used as the map solution for the project. Also, the essential part for the thesis project is deck.gl support for the point cloud layer that shows every piece of data as a 3D sphere on a certain position. (Uber 2020c.)

2.4 Point cloud

Provided data by the SILAM software may seem hard to process if to look on it one by one value, but altogether it forms a logical shape. Resulting data is represented in a grid system in 4 dimensions: longitude, latitude, altitude and time. Values for dimensions are the same for every dispersed material, only the actual material's concentration value is different. Resulting data can be illustrated as points lying on a concrete coordinate in space. This approach better correlates with the point cloud technique.

A point cloud is representing the number of data points forming figures in space. This approach for data visualizing gained huge popularity for lidar systems. The lidar system rotates a laser on high speed and scans it surrounding resulting in a 3D picture of space around. Nowadays it is heavily used by autonomous driving vehicles and allows them to better understand the environment around them. For example, Figure 6 shows how the car sees its surrounding with the lidar system alongside with camera output. Objects on the map are distinguished and correlate to camera output, but everything on the map is represented as a set of

points forming end objects. Also, this example is built with Mapbox GL JS and deck.gl. (Uber 2020.)

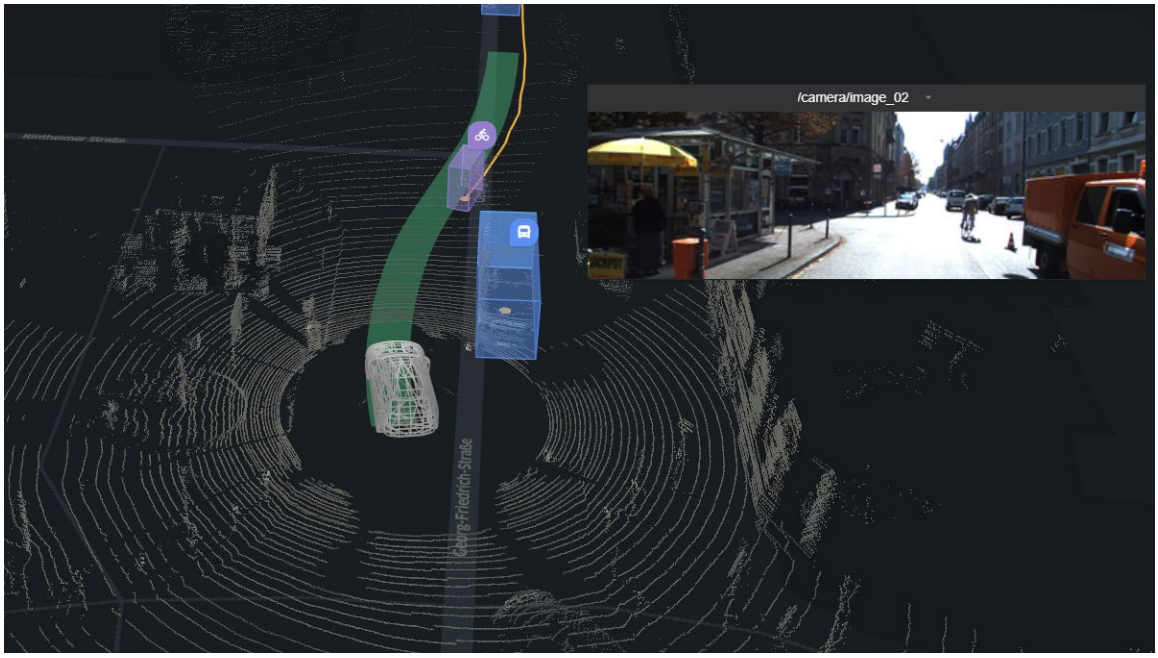


Figure 6. Autonomous Visualization System (AVS) demo (Uber 2020)

Moreover, the point cloud is great for comparing data values. Data difference, for example, can be illustrated with the point colour, transparency or size. Figure 7 shows the point cloud representation of a tea pod with the colour map where higher points are yellow and lower points are violet.

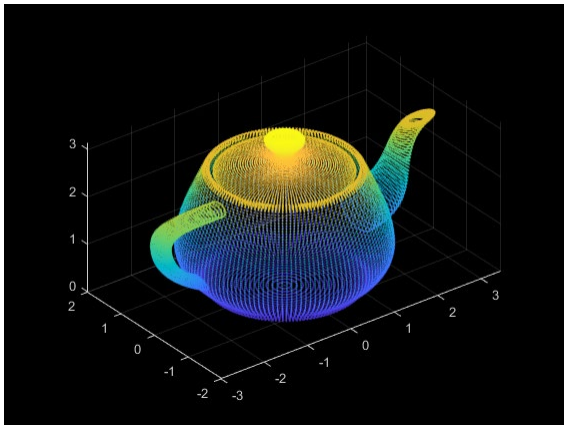


Figure 7. 3D point cloud of tea pod (MathWorks 2020)

The main benefit of this approach for the thesis project is that SILAM data can be easily represented as a cloud point layer. As it was mentioned, the data about every material is stored as numerical values of the material's concentration on a set of the coordinates. This set of coordinates is saved in a grid form and it is the

same for every material, while only the material's concentration is different. Every concentration value can be represented as a single sphere point in the space on predefined coordinates. The concentration value can be shown, for example, with point colour or point size. Finally, all these rendered spheres will take a form of the cloud, like points in Figure 7 take a form of the tea pod. This technique will make every piece of important data to be actually used and to be visible with tiny 3D spheres forming a real size cloud in the air. It will improve the user experience by allowing a user to fly over the cloud, zoom in to the specific area and finally make the right choice in a critical situation.

3 PROJECT IMPLEMENTATION

This chapter will cover the implementation of the project application. At first, in Section 3.1 the planned application will be described. After that, in Section 3.2 the project setup will be explained. Next, Section 3.3 and Section 3.4 will be about visualization tools integration. Description and explanation of the provided data will be given in detail in Section 3.5. Following section will be about data handling and algorithms for processing provided data. The result of the processed data will be shown in Section 3.8. An attempt at generating more data will be described in Section 3.9. The result of generated dispersed clouds will be shown in Section 3.10. Finally, Section 3.11 will compare the resulted visualization to the current solution by the commissioner company.

3.1 Application planning

The initial idea is directly referred to as the dispersed cloud implementation on the map in Observis' SAS software. Resulting data from the SILAM system is represented as a heatmap on the map. Heatmap is a 2D graphical representation of data, where every data value is shown with different colour depending on the maximum value. In this case, the user does not see the whole cloud on the map but its rough 2D representation from the above. Figure 8 shows how SILAM data is represented in the SAS software. There is a 2D layer of the cloud in the centre and the control centre on the bottom of the picture. The layer is a completely flat rendered figure which shows the concentration of the material on some height.

The control centre is defined as a set of six buttons for changing the materials, time period and height of the layer.

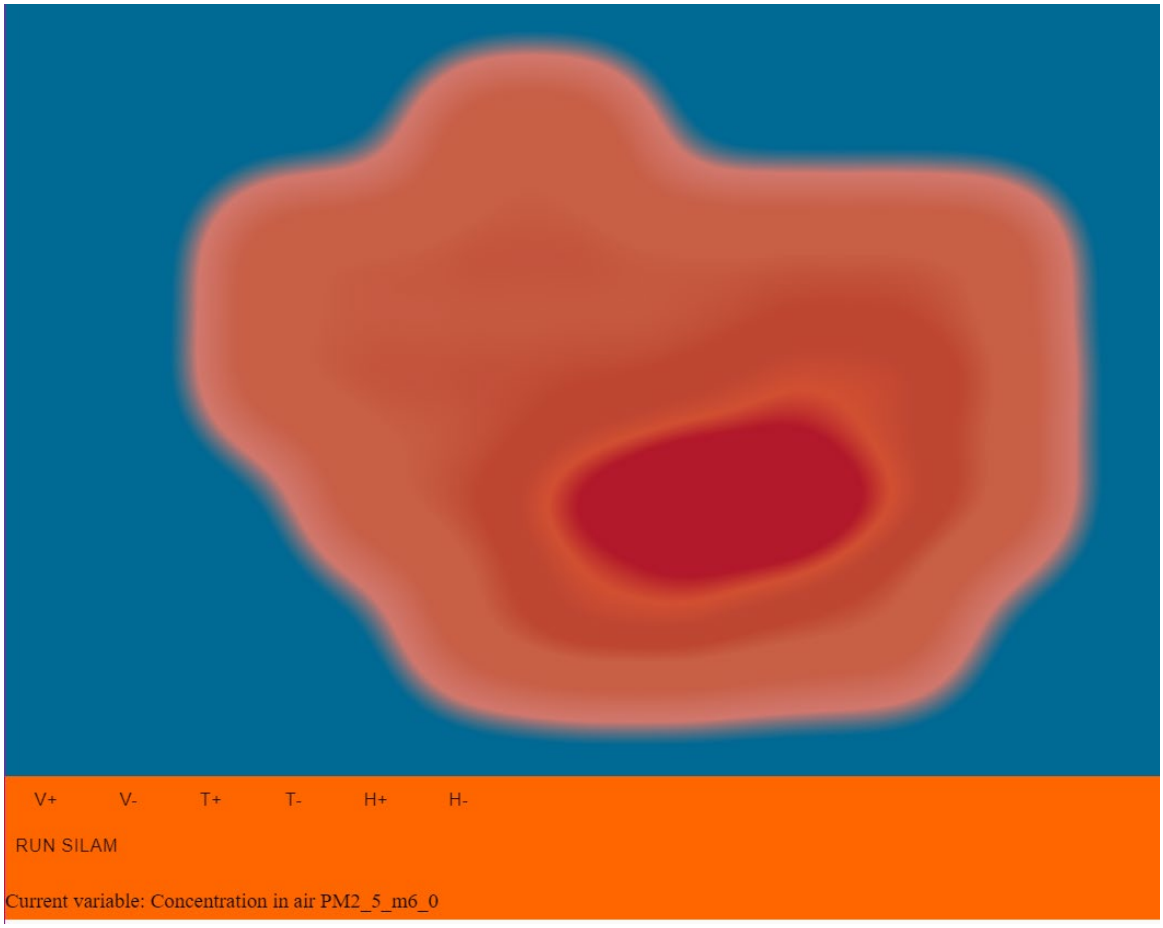


Figure 8. The SILAM data heatmap

In Observis' solution the cloud is horizontally divided into many layers represented as different altitudes. By tapping a button on a screen a user can change the layers and see how data shifts from one layer to another. This approach has these problems as follows:

- 2D layers. It is hard to quickly and correctly estimate the whole situation. There is a need to switch for every layer to get the whole picture of the cloud.
- Extra actions. It is impossible to see the whole cloud. A user has to switch from layer to layer by performing unwanted actions.
- Lack of interaction. Cloud layers were static rendered graphics that show data only in a selected hour without any animation or transition.

These problems led to rethinking the current solution of Observis. At first, it was decided that cloud representation will be in full 3D. In this way, the user will see the whole cloud instantly and situation analysis will be drastically reduced in time.

Also, it frees customers from extra actions by changing cloud layers for every height value. Next, a good idea would be to add a new feature by animating the cloud in time. In this way, the cloud would change shape automatically showing the cloud during available hours. With this in mind, it was decided to create a web app with a map in the centre and the control centre in the screen corner. The map will display the generated cloud from SILAM data as a cloud of points. Points will have different colours depending on their value. The control centre should include:

- Gas data change as a dropdown list
- Hour slider
- Animation toggle

The application will be introduced not as a ready to ship product. It will be a proof-of-concept application trying to solve the current solution's problems. It involves new ideas related to 3D graphics that were not implemented by Obsevis' SAS software competitors. As any new creation, it will show another idea for showing a rendered cloud on the map and introduce new possibilities and problems. Next chapters describe the journey of creating the thesis project application and give answers on questions about the project structure, UI setup and data processing. Also, show the result of an attempt to enhance visualization by generating more data. Finally, the created application will be demonstrated.

3.2 Project setup

This section covers the architecture of the application. It describes the way of integrating the main technologies. Also, it shows how the source code is divided into different files correspondingly to the Electron runtime logic.

The project will be created with the Visual Studio Code text editor by Microsoft. It supports syntax highlight for programming languages used in the project, has an integrated terminal for easy code execution and advanced flexibility in editor setup. In the beginning of the project a boilerplate project will be used that already has all core components like Electron, React and TypeScript integrated and working. Also, it includes a Webpack component that can update the content

of running an application on the fly at code change and bundle all the code to one package. All this is aimed to accelerate development speed and gives more time to concentrate on project goals. The project folder structure is presented in Figure 9 consists of several folders and configuration files.

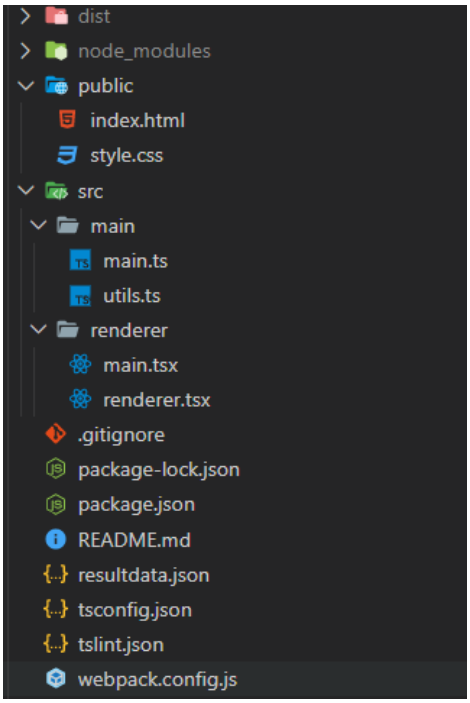


Figure 9. Project folder structure

The first folder is **dist.** It contains a compiled project and on every compilation its files are updated to the latest code changes. All imported modules and their dependencies are stored in **node_modules**. Folders **public** and **src** are very common in web-based applications. In **public** an html base file is stored that imports all the content from the **src** folder. More precisely, the **src** folder has all the application logic and UI that is exported to the html base file. Folders **main** and **renderer** inside **src** are common for Electron applications. The application main logic is usually stored inside **main**. It is aimed to application windows rendering, windows lifecycle managing and preparing content for UI. Also, the root folder has the file **resultdata.json** which is a SILAM output data file.

Additionally, it is essential to cover how Webpack performs. The Webpack main function is to combine all separate code files and assets to several files in smaller size and quantity. Webpack runs its own server which looks for code changes and on every “Save” action it recompiles the project. Moreover, Webpack has an

intelligent packing system. It automatically detects imported third-party libraries and does not adds the whole library, but includes to bundle only the used components of that library. In this way, the code contains only the needed imported packages reducing the size of the content which needs to be shipped to a potential user. Figure 10 shows the piece of code responsible for bundling source code.

```
const mainConfig = lodash.cloneDeep(commonConfig);
mainConfig.entry = './src/main/main.ts';
mainConfig.target = 'electron-main';
mainConfig.output.filename = 'main.bundle.js';
mainConfig.plugins = [
  new CopyPkgJsonPlugin({
    remove: ['scripts', 'devDependencies', 'build'],
    replace: {
      main: './main.bundle.js',
      scripts: { start: 'electron ./main.bundle.js' },
      postinstall: 'electron-builder install-app-deps',
    },
  }),
];

const rendererConfig = lodash.cloneDeep(commonConfig);
rendererConfig.entry = './src/renderer/renderer.tsx';
rendererConfig.target = 'electron-renderer';
rendererConfig.output.filename = 'renderer.bundle.js';
rendererConfig.plugins = [
  new HtmlWebpackPlugin({
    template: path.resolve(__dirname, './public/index.html'),
  }),
];

module.exports = [mainConfig, rendererConfig];
```

Figure 10. Webpack bundling configuration (webpack.config.js)

These instructions include target files and output files, compiler targets and additional plugins. For example, in the second chunk of code from Figure 10 “HtmlWebpackPlugin” plugin is used. It takes the html base file **public/index.html** as a template to automatically generate links to all the assets for final UI rendering. Finally, generated files are saved in the **dist** folder and take much less space than the whole project.

3.3 Preparation of visualization tools

This section introduces the way of adding the external libraries to the project. More precisely, the process of integrating visualization libraries is described here. Also, it shows the minor problem of working with TypeScript, when there are no official typing modules for the desired libraries.

To start working with deck.gl and Mapbox they must be imported to the project. For adding these libraries script tool **npm** is used. It is one of the command line package managers used for creating a web-based application. It can install any available package with its dependencies from open repositories. Every imported package and its dependencies are saved to the **node_modules** folder. For example, for installing deck.gl the command “npm install deck.gl” was used.

Instead of using plain Mapbox GL JS package preferences were given to the Mapbox wrapper from Uber called “react-map-gl”. Both deck.gl and react-map-gl are from the same developer and have better integration with each other. react-map-gl is the same Mapbox GL JS library with an extra React component wrapper. This system better performs rendering 2D and 3D visualization graphics on the map. It adds an extra layer holding all rendered visualization that overlays the actual map and synchronizes the camera movements. In this approach, visualization is rendered independently from the map and can use its own optimized algorithms. react-map-gl is installed with the very similar command “npm install react-map-gl”.

To make deck.gl and react-map-gl operational in the project environment extra steps are required. TypeScript needs declaration files for making libraries reachable to it. They are installed in the same way as libraries, just with **npm** commands. In the time of writing this thesis deck.gl has no official global typing module. In this case, a module created by deck.gl development community was used. However, the use of non-official libraries can lead to less reliable performance and poor support. For example, in this module, a bug related to the layer coordinate system was fixed during the application development. Installation of this module is performed with the “npm install

@danmarshall/deckgl-typings" command. react-map-gl on other hand has official typing modules and its installation is done with the "npm install @types/react-map-gl" command.

Finally, all the needed modules are imported and visualization tools are ready for the integration. The next chapter will be about building a frontend part and visualization tools setup.

3.4 Frontend setup

After integrating all the visualization libraries, they become operational in the project. This section shows how visualization tools are arranged and how they integrate into the application user interface. Also, there is a description of properties which are needed for displaying the cloud point layer. It gives an understanding of how the data should be processed in the backend for the later easy use by visualization tools.

The frontend for the project is built completely with React. Nevertheless, the application has the html base file **public/index.html** that will render all the interface inside a single div element. Figure 11 shows its code and div element with the id "app" which will be used by a parent React component to link with it.

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Mikhail</title>
</head>

<body>
  <div id="app" class="app"></div>
</body>

</html>
```

Figure 11. Base HTML file (public/index.html)

Code for UI rendering is placed in the **src/renderer** folder. It has two files **main.tsx** and **renderer.tsx**. The extension "tsx" of these files indicates that both are React files with code written in TypeScript language. The file **renderer.tsx** is a starting point for UI code. It holds the initial code that renders React element to

DOM. The following Figure 12, shows a “ReactDOM.render()” function from the React package that injects a rendered component inside the supplied container. In this case, it renders a “div” element with another React element inside called “Main” imported from the **src/renderer/main.tsx** file. After that, a renderer “div” with all its content is inserted to the element with the id “app” inside the **public/index.html** base html file.

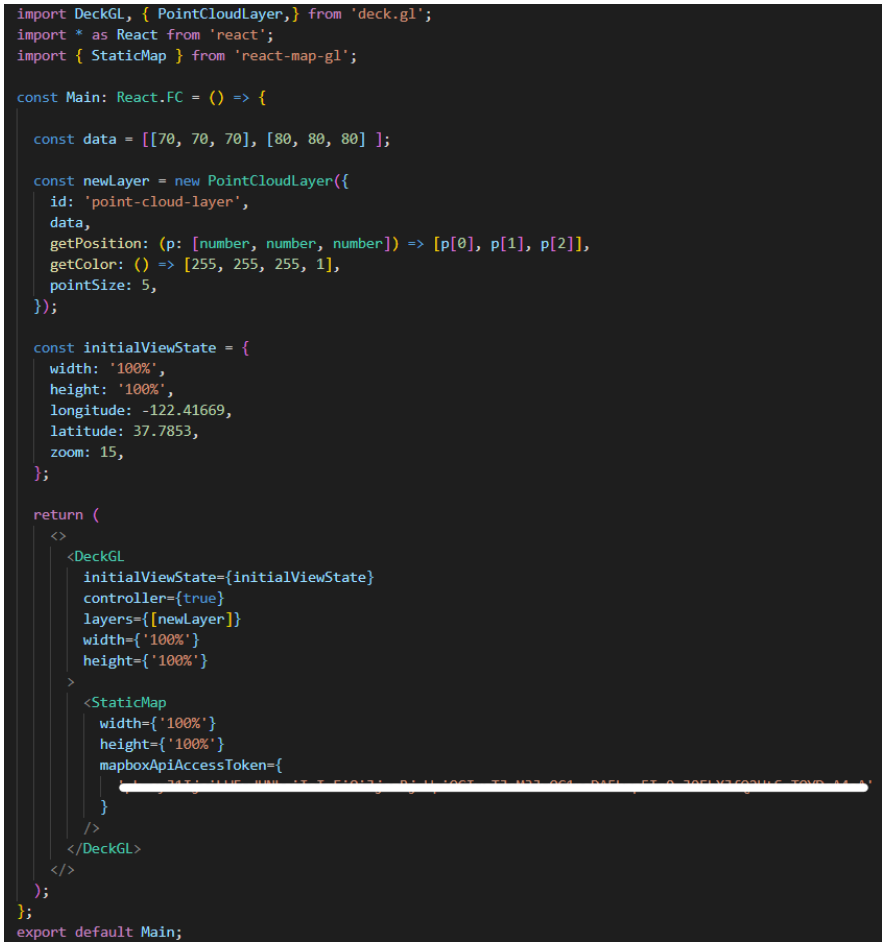
```
import * as React from 'react';
import * as ReactDOM from 'react-dom';
import Main from './main';
import '@public/style.css';

ReactDOM.render(
  <div className='app' style={{display: 'flex'}}>
    <Main/>
  </div>,
  document.getElementById('app')
);
```

Figure 12. Starting point for React rendered UI (src/renderer/renderer.tsx)

The biggest part of code for UI rendering is held in **main.tsx**. It has a functional React component **Main** performing the most of UI rendering instructions. The following Figure 13 shows how little needs to be configured for running visualization tools. React component **Main** renders the Mapbox map and the deck.gl layer on top of it. To make the map operational it needs an API token which is given after registration on Mapbox website. The point cloud layer from the deck.gl package has some following properties used in the project:

- **id** – unique layer name, which in this case, can be anything as only one layer will be drawn
- **data** – an array of data used for point drawing
- **getPosition()** – the function that iterates through **data** and returns retrieved coordinates in number array format with the order: longitude, latitude, altitude
- **getColor()** – a function that iterates through **data** and returns retrieved point colour in array format with RGBA value format
- **pointSize** – setting radius for all points in pixel format by default

Also, the “DeckGL” component has the property “initialViewState” holding a layer size and default values for view positioning.



```
import DeckGL, { PointCloudLayer,} from 'deck.gl';
import * as React from 'react';
import { StaticMap } from 'react-map-gl';

const Main: React.FC = () => {

  const data = [[70, 70, 70], [80, 80, 80] ];

  const newLayer = new PointCloudLayer({
    id: 'point-cloud-layer',
    data,
    getPosition: (p: [number, number, number]) => [p[0], p[1], p[2]],
    getColor: () => [255, 255, 255, 1],
    pointSize: 5,
  });

  const initialViewState = {
    width: '100%',
    height: '100%',
    longitude: -122.41669,
    latitude: 37.7853,
    zoom: 15,
  };

  return (
    <>
      <DeckGL
        initialViewState={initialViewState}
        controller={true}
        layers={[newLayer]}
        width={'100%'}
        height={'100%'}
      >
        <StaticMap
          width={'100%'}
          height={'100%'}
          mapboxApiAccessToken={{
          }
        />
      </DeckGL>
    </>
  );
};
export default Main;
```

Figure 13. The required minimum for rendering visualization tools (src/renderer/main.tsx)

The execution of the Electron application is handled by **npm** commands. For web-based applications they are usually located inside a **package.json** configuration file. They can be fully modified for application needs. The execution of this application is performed by two commands, one after another: “npm run dev” and “npm start” in Figure 14. The first command runs the Webpack server that compiles the application and puts the compiled code to the **dist** folder. The second command reads the **dist** folder content and opens a window of the compiled Electron application.

```
"scripts": {
    "start": "electron ./dist/main.bundle.js",
    "dev": "rimraf dist && cross-env NODE_ENV=development webpack --watch --progress --colors",
    "prod": "rimraf dist && cross-env NODE_ENV=production webpack --progress --colors",
    "test": "mocha -r ts-node/register -r tsconfig-paths/register \"test/**/*.ts\"",
    "build:win": "electron-builder build --win --x64",
    "build:mac": "electron-builder build --mac --x64"
  },
```

Figure 14. Project npm scripts (package.json)

Executing written UI code will result in a Mapbox map and a point cloud layer, both on set coordinates. Figure 15 demonstrates the executed application's window after visualization tools initial setup.

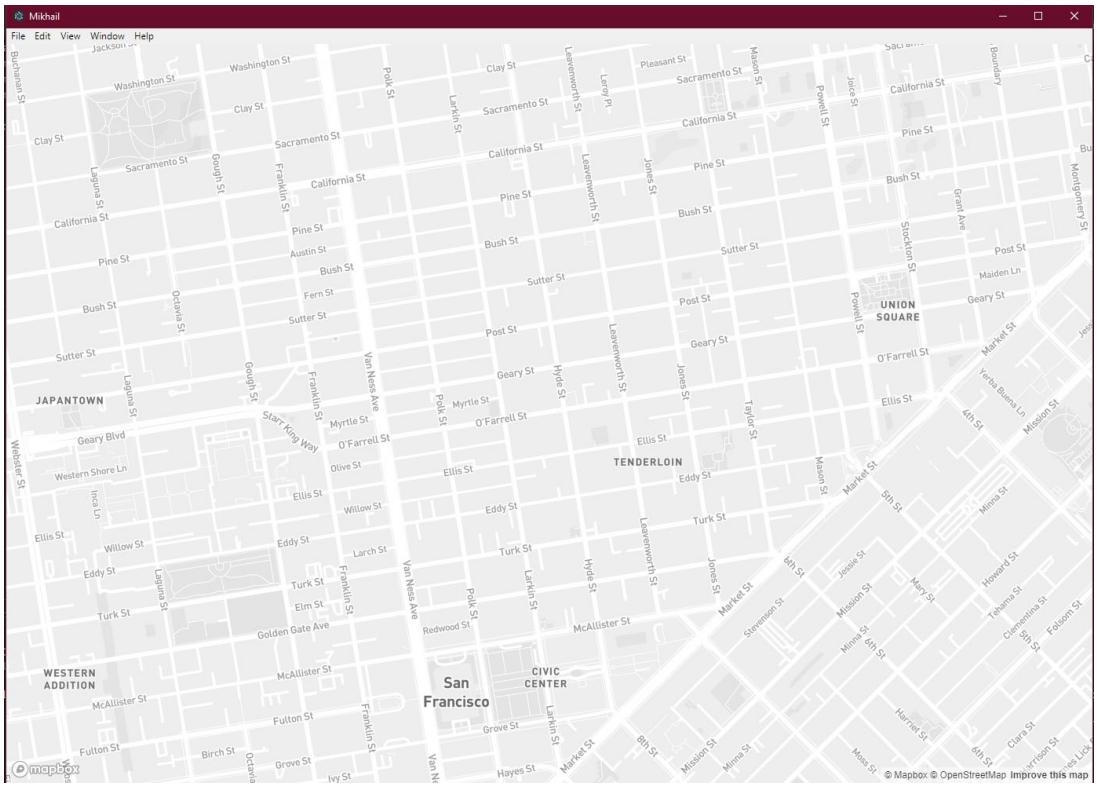


Figure 15. Application after initial visualization tools setup

After understanding how visualization works in the project, it is time for observing SILAM output data. The next section will describe how SILAM data is organized and typed.

3.5 Understanding SILAM data

When the frontend is ready for applying the data, the next step is to process the data appropriately. However, before designing the data processing logic, it is an essential step to understand how the raw data is organized. This section will explain how the SILAM output data is written, which format is used and how the data is actually stored.

Initially, SILAM output data is written in the netCDF format that is an open-sourced format for exchanging multi-dimensional scientific data. However, reading this format can be an extra task during programming. To make it easier

for understanding and processing, data have been formatted to JSON with the Linux script ncsk. After described instructions data have been given from the commission company for the thesis project.

JSON is a popular format for data distribution among software. It keeps all data in nested key-value pairs format making it human-readable and easy to process. Respectively, SILAM data is divided into key-value pairs, too. Root keys are **dimensions**, **variables**, **attributes**. Figure 16 shows the diagram of the main keys and the data saved under them.

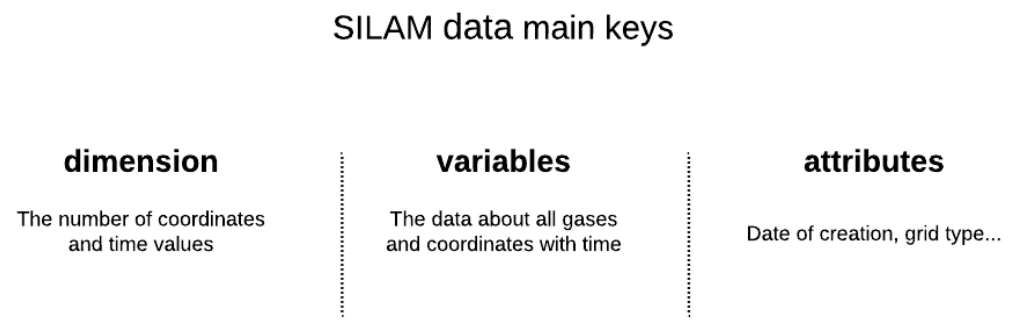


Figure 16. Main keys in the SILAM source file

All the important data is stored under **variables** key. Nevertheless, it is worth to mention the rest. **Attributes** key keeps data about SILAM version, model creation date and grid type. In this case, data about dispersed material is stored in a grid type “lonlat” which is longitude and latitude. Longitude is a geographic coordinate specifies a east-west position, while latitude specifies a north-south position. Also, SILAM includes altitude data which is known as a height coordinate. On a flat surface like world map, they are perpendicular to each other forming a grid system or coordinate system. With this knowledge, SILAM software writes data in a grid system where for each longitude-latitude-altitude coordinate there is some numeric value for every dispersed material. Nevertheless, grid type called “lonlat” because longitude and latitude values difference is the same within its coordinates, while height values difference is different from value to value. Coordinate data is stored under **variables** key in the format illustrated in Figure 17. Key “data” keeps coordinate values for every three coordinates.

```
"height": {
  "shape": ["height"],
  "type": "float",
  "attributes": {
    "units": "m",
    "positive": "up",
    "long_name": "layer midpoint constant height from surface",
    "axis": "Z",
    "standard_name": "layer_midpoint_height_above_ground"
  },
  "data": [12.50, 50.0, 125.0, 275.0, 575.0, 1150.0, 2125.0, 3725.0, 5725.0]
},
"lat": {
  "shape": ["lat"],
  "type": "float",
  "attributes": {
    "units": "degrees_north",
    "axis": "Y",
    "long_name": "latitude",
    "standard_name": "latitude",
    "_CoordinateAxisType": "Lat"
  },
  "data": [60.0, 60.0250, 60.050, 60.0750, 60.10, 60.1250, 60.150, 60.1750, 60.20,
},
"lon": {
  "shape": ["lon"],
  "type": "float",
  "attributes": {
    "units": "degrees_east",
    "axis": "X",
    "long_name": "longitude",
    "standard_name": "longitude",
    "_CoordinateAxisType": "Lon"
  },
  "data": [-10.0, -9.950, -9.90, -9.850, -9.80, -9.750, -9.70, -9.650, -9.60, -9.55
},
```


Figure 17. SILAM coordinate data (resultdata.json)

Dimensions hold the number of values for each dimension coordinate and time. Time data is stored under the **variables** key in the same way as dimension coordinates. Dimensions data is shown in Figure 18.


```
"dimensions": {
  "lon": 200,
  "lat": 400,
  "height": 9,
  "time": 6
},
```


Figure 18. SILAM dimensions list (resultdata.json)

All important data is placed under the **variables** key. It is not only coordinates and time but all the dispersed materials information. Figure 19 illustrates a piece of data from the **variables** key. On the picture information about three dispersed materials is shown: “U_wind_10m”, “V_wind_10m” and “cnc_PM2_5_m6_0”. Each one has its data under keys:

- **shape** string array describes the **data** value nested array, indicates in which dimension and time the data is written
- **type** data type of array under the **data** key
- **attributes** material's additional information
- **data** nested array of material's values, nesting order is defined in value array of **shape** key

Also, some of the materials do not have information about height. For example, in Figure 19 materials "U_wind_10m" and "V_wind_10m" do not have a record "height" in the array under the **shape** key. Moreover, the value of the key **data** is a 3-dimensional array which corresponds to 3 records in the **shape** array "["time", "lat", "lon"]". On another hand, material "cnc_PM2_5_m6_0" has a record "height" in its **shape** array. In this case, the value of key **data** is a 4-dimensional array which corresponds to 4 records in the **shape** array " ["time", "height", "lat", "lon"] ".


```
"variables": {
  "U_wind_10m": {
    "shape": ["time", "lat", "lon"],
    "type": "float",
    "attributes": {
      "_FillValue": -9.999990e+14,
      "units": "",
      "long_name": "U-component of 10m wind [m/s]"
    },
    "data": [[[-1.765353, -1.566085, -1.378445, -1.219790, -1.057693, -0.9427836, -0.8636648, -0.
  },
  "V_wind_10m": {
    "shape": ["time", "lat", "lon"],
    "type": "float",
    "attributes": {
      "_FillValue": -9.999990e+14,
      "units": "",
      "long_name": "V-component of 10m wind [m/s]"
    },
    "data": [[[9.848985, 9.901314, 9.940005, 9.946832, 9.953028, 9.962452, 9.976740, 9.995852, 10
  },
  "cnc_PM2_5_m6_0": {
    "shape": ["time", "height", "lat", "lon"],
    "type": "float",
    "attributes": {
      "_FillValue": -9.999990e+14,
      "substance_name": "PM2_5",
      "silam_amount_unit": "kg",
      "mode_name": "",
      "mode_distribution_type": "FIXED_DIAMETER",
      "mode_nominal_diameter": "6.0000000            um",
      "fix_diam_mode_min_diameter": "2.5000000            um",
      "fix_diam_mode_max_diameter": "10.0000000           um",
      "fix_diam_mode_mean_diameter": "6.0000000            um",
      "units": "kg/m3",
      "long_name": "Concentration in air PM2_5_m6_0"
    },
    "data": [[[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
  },
```


Figure 19. SILAM data under "variables" key (resultdata.json)

Figure 20 shows how the material's concentration data can be unwrapped from the multidimensional array. The data inside the array is encapsulated in a way

like a matryoshka doll has smaller dolls inside a bigger doll until the smallest doll is out. In this case, every “doll” represents the coordinate or time value until the smallest doll is reached which is the concentration value. The array "shape" defines the order of the encapsulation and helps to understand how deep the unwrapping algorithm needs to go.

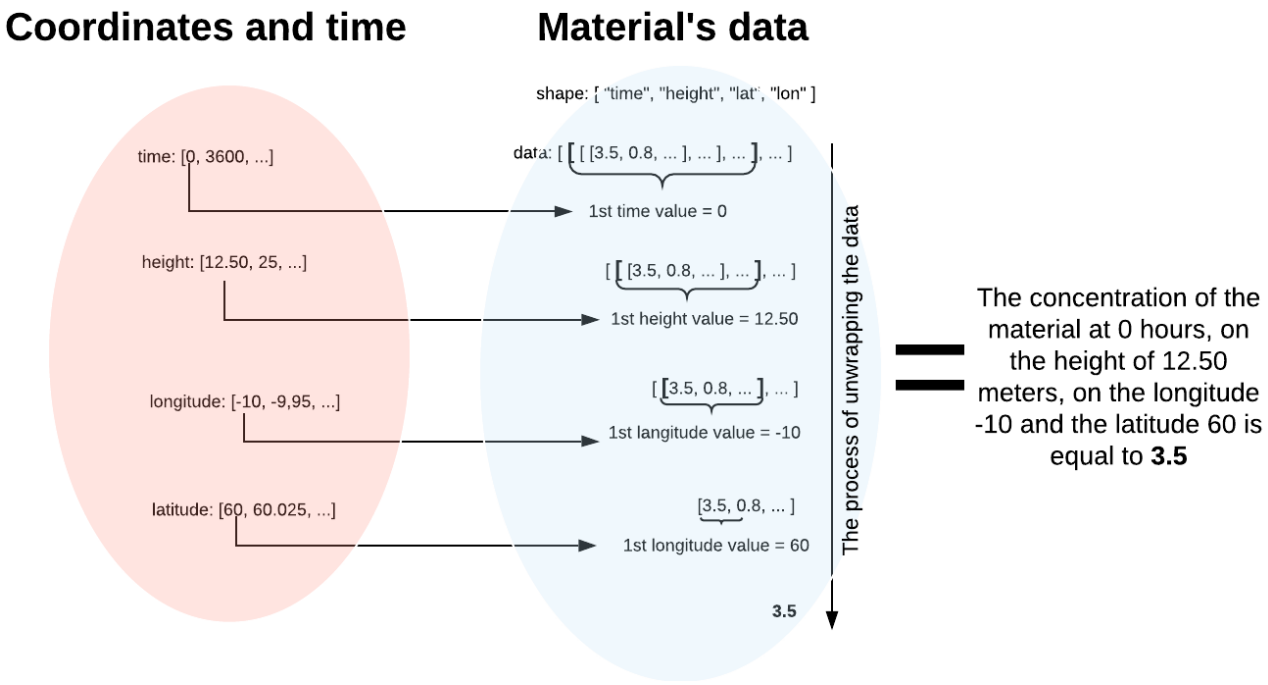


Figure 20. Unwrapping the SILAM data

From this information, it is now possible to estimate what data will be needed for correct visualization. For drawing the cloud point layer important information is the point coordinate for correct placement and the value for calculating its colour. Next chapter will cover data processing algorithms.

3.6 Data processing

This important section will be dedicated to data processing. After understanding how raw data is organized, the first step is to set the types for the desired data. It helps to clarify what data is really essential for the point cloud layer. It is a very important part because the data types form the whole architecture of how data will be processed and stored inside the application memory. After defining the data types, the second step is to implement the algorithm for data processing. It will apply the same logic described in Figure 20. It will unwrap the

multidimensional array for retrieving the most important pieces of data like coordinates, time value and the concentration value.

It is a good habit to logically separate source code to different files. With this motivation, a separate file **src/main/utils.ts** was created for code related to processing data, while file **src/main/main.ts** contains only Electron application code. It helps divide code to smaller modules that easier to maintain and debug. Also, it is faster to locate a malfunction and fix it without touching unrelated code.

Data processing algorithm will read the value of the key **variables** from the SILAM output data file. Specifically, longitude, latitude, height and time values will be used. Then for every material, the data will be taken from keys **data**, **shape** and **attributes.long_name**. This is enough data to make the point cloud layer. Moreover, this choice of data is trying to minimize the processing of unwanted data like all **attributes** values to reduce the amount of processed data and execution time.

3.6.1 Defining data types

First thing is to define data types used for reading SILAM data. It will help to organize a process of raw data and later use of end data. This subsection will explain each created data type and describe how they will be used.

The first is type **Variables** in Figure 21, it is used for processing only needed data from the file “resultdata.json”. It includes every material and information about dimensions and time. It will be not an end data because for a deck.gl point cloud layer it needs to be an array of objects with coordinates and colour values. **Variables**, on the other hand, stores coordinates and values in separate locations. Moreover, materials’ value data is still in the multidimensional array which makes it hard to read.

```
export type Variables = {
  [key: string]: {
    data: number[][][][];
    shape: string[];
    attributes: { long_name: string };
  };
} & {
  [key: string]: {
    data: number[][][];
    shape: string[];
    attributes: { long_name: string };
  };
} & {
  lon: { data: number[] };
  lat: { data: number[] };
  time: { data: number[] };
  height: { data: number[] };
};
```

Figure 21. Type Variables (src/main/utils.ts)

Then, interface **GasNames** in Figure 22 is used for keeping a list of all available materials read from raw data in type **Variables**.

```
export interface GasName {
  name: string;
  key: string;
}
```

Figure 22. Interface GasNames (src/main/utils.ts)

After, interface **Point** in Figure 23 is used for saving every individual point with value, colour and coordinates. An array of this type forms completely ready data for the deck.gl point cloud layer as it has all important information to draw the layer.

```
interface Point {
  position: { latitude: number; longitude: number; altitude: number };
  value: number;
  color?: RGBAColor;
}
```

Figure 23. Interface Point (src/main/utils.ts)

Next, interface **ResultData** in Figure 24 saves an array of data points in a specific point of time with the material's maximum value. As provided SILAM data has information about 6 hours (see Figure 16), it will be 6 variables of **ResultData** for every time period and with an absolutely different array of points.

```
export interface ResultData {
  data?: Point[];
  maxValue: number;
  time: number;
}
```

Figure 24. Interface ResultData (src/main/utils.ts)

Finally, class **Result** in Figure 25 is gathering all essential information about dispersed material: its name, array of data point for different periods and nested dictionary of all used coordinates. Enum **DataKeys** is supplementary, it used in coordinates' nested dictionary inside **Result** to indicate the order of nesting and acts as dictionary keys for coordinate data.

```
export enum DataKeys {
  alt = 'altitude',
  t = 'time',
  lat = 'latitude',
  lon = 'longitude',
}

export class Result {
  data: ResultData[];
  name: string;
  coordinates: {
    keys: [DataKeys.t, DataKeys.alt, DataKeys.lat, DataKeys.lon];
    data: { [key: number]: { [key: number]: { [key: number]: number[] } } };
  };

  constructor(
    data: ResultData[],
    name: string,
    cdata: { [key: number]: { [key: number]: { [key: number]: number[] } } },
  ) {
    this.data = data;
    this.name = name;
    this.coordinates = {
      keys: [DataKeys.t, DataKeys.alt, DataKeys.lat, DataKeys.lon],
      data: cdata,
    };
  }
}
```

Figure 25. Class Result and enum DataKeys (src/main/utils.ts)

Class **Result** defines fully processed data for a dispersed material. Next step covers how data will be processed using described custom-made data types.

3.6.2 Data processing functions

Data processing algorithms will be divided into functions. With this approach, it is easier to maintain them and remember their functionality after some time. They

will be created in the same file as custom-made data types, inside the file **src/main/utils.ts**.

The first step before processing the data is to read it and to save it into the application memory. The function of Figure 26 is designed for that process. It reads the data from the “resultdata.json” file and returns it in the object with the data type **Variables.**

```
export function getSilamDataFromJson(): { variables: Variables } {
  const dataset = require('../../resultdata.json');
  const loadData = () => JSON.parse(JSON.stringify(dataset));
  const data: { variables: Variables } = loadData();
  return data;
}
```

Figure 26. Function for reading SILAM data (src/main/utils.ts)

When data is accessible from the program’s memory, the next step is to work with it. Figure 27 shows a function that returns a name list of all available dispersed materials from the data saved to the application memory. To name materials which have information about height three asterisk characters are added. Later, this compiled list will be used for selecting the desired material to view its data.

```
export function getGasNames(dataset: { variables: Variables }): GasName[] {
  const variables = dataset.variables;

  const gasNames: GasName[] = [];
  const gases = Object.keys(variables).filter(
    gasName => !['height', 'time', 'lon', 'lat'].toString().includes(gasName),
  );
  gases.forEach(gas => {
    gasNames.push({
      key: gas,
      name: variables[gas].shape.includes('height') ? `***  ${gas}` : gas,
    });
  });
  return gasNames;
}
```

Figure 27. Function for retrieving a list of all dispersed materials (src/main/utils.ts)

The most important and responsible task is to get all the needed data for a selected dispersed material and prepare it right for the point cloud layer. Values’ data for every material is stored in a multidimensional array. Figure 28 shows the first half of the function that unravels a multidimensional array and generates

points' data out of it. The second half of the function is doing the same thing except not reading height information, because it is not present for some materials. The function takes two parameters: dispersed materials' name and raw SILAM data. It reads a multidimensional data array and for every value creates a point with value and its coordinates. Also, it takes record of the maximum value for generating the colour of the point later. At the end of the function, colour is set for every point with another function. Finally, out of all the points made with coordinates and values, this function returns the object type **Result**.

```
export function getGasData(
  dataset: { variables: Variables },
  gasName: string,
): Result {
  let data: Point[] | undefined;
  let results: ResultData[] | undefined;
  let maxValue: number = 0;
  let time: number;
  let latitude: number;
  let longitude: number;
  let altitude: number;
  let longitudes: number[] = [];
  let latitudes: { [key: number]: number[] } = {};
  let altitudes: { [key: number]: { [key: number]: number[] } } = {};
  let time_coord: {
    [key: number]: { [key: number]: { [key: number]: number[] } };
  } = {};

  const variables = dataset.variables;

  if (variables[gasName].shape.includes('height')) {
    for (let t = 0; t < variables[gasName].data.length; t++) {
      for (let h = 0; h < variables[gasName].data[t].length; h++) {
        for (let lat = 0; lat < variables[gasName].data[t][h].length; lat++) {
          for ( let lon = 0; lon < variables[gasName].data[t][h][lat].length; lon++ ) {
            const value = variables[gasName].data[t][h][lat][lon];
            if (value > 0) {
              maxValue = value > maxValue ? value : maxValue;
              latitude = variables.lat.data[lat];
              longitude = variables.lon.data[lon];
              altitude = variables.height.data[h];

              if (data) {
                data.push({value, position: { latitude, longitude, altitude,},});
              } else {
                data = [{value,position: { latitude, longitude, altitude, },},];
              }

              longitudes.push(longitude);
            }
          }
        }
      }
      time = variables.time.data[t];

      if (results) {
        results.push({ maxValue, time, data: data! });
      } else {
        results = [{ maxValue, time, data: data! }];
      }
      data = undefined;
    }
```

Figure 28. Function for generating point data out of SILAM data multidimensional array (src/main/utils.ts)

The functions in Figure 29 are used when data points are generated for applying colour value to them. The second function reads the point's value and maximum value for colour and opacity calculation based on the difference from a point's value to the maximum value. Closer value to the maximum value, the redder it will be, and vice versa. The smaller the value relative to the maximum value, the greener it will be. Opacity has four levels, the smaller the value relative to the maximum value, the more transparent the point will be.

```
export function generateColorForPoints(result: Result) {
  result.data.forEach(data => {
    const maxValue = data.maxValue;
    data.data &&
      data.data.forEach(point => {
        point.color = getColorForPoint(point.value, maxValue);
      });
  });

  return result;
}

function getColorForPoint(value: number, maxValue: number): RGBAColor {
  const diff = value / maxValue;
  const opacity = Math.ceil(diff / 0.25);
  const color: RGBAColor = [
    diff >= 0.5 ? 255 : diff * 2 * 255,
    diff <= 0.5 ? 255 : (1 - diff) * 255,
    1,
    255 * opacity * 0.25,
  ];
  return color;
}
```

Figure 29. Functions for generating colour for passed points (src/main/utils.ts)

Finally, these functions with custom made data types are ready for reaching all the needed information from SILAM data about any available dispersed material and prepare it for visualization. The next part will describe a workflow of passing ready data to the visual interface.

3.7 Data transfer

When the processed data is ready, the next step is to visualize it. At first, the data needs to be transferred from the backend straight to the frontend. However, due to specifics of the Electron environment, the communication between two application's sides is organized by an integrated API. It is needed because the Electron application runs two separate processes: one for the application logic,

another one for UI rendering. Their communication is set by pairs of listening and triggering functions on both frontend and backend side. This section describes how to set the communication channels and pass the data between two main Electron processes.

The architecture of Electron applications is built on top of two types of processes. The **main** process is the starting point for a whole application. It initializes the application and calls the GUI to render the content. An Electron application runs only one **main** process.

Rendering the UI, on the other hand, is handled by other processes. Since Electron uses Chromium, displaying the UI is made with multi-process architecture. Each window for an Electron application runs its own **renderer** process.

Data transfer between two separate processes can be a problem, but not for an Electron application. The Electron development team has created a special API with two main modules for this kind of communication: **ipcRenderer** and **ipcMain**. Figure 30 illustrates how asynchronous communication takes place. Every time communication starts from the renderer process. At first, **ipcRender.send()** triggers a listener **ipcMain.on()** by passing a string channel value. Then, **ipcMain** sends data back to the renderer process with another channel. The renderer listens to this channel by **ipcRenderer.on()** and retrieves passed data.

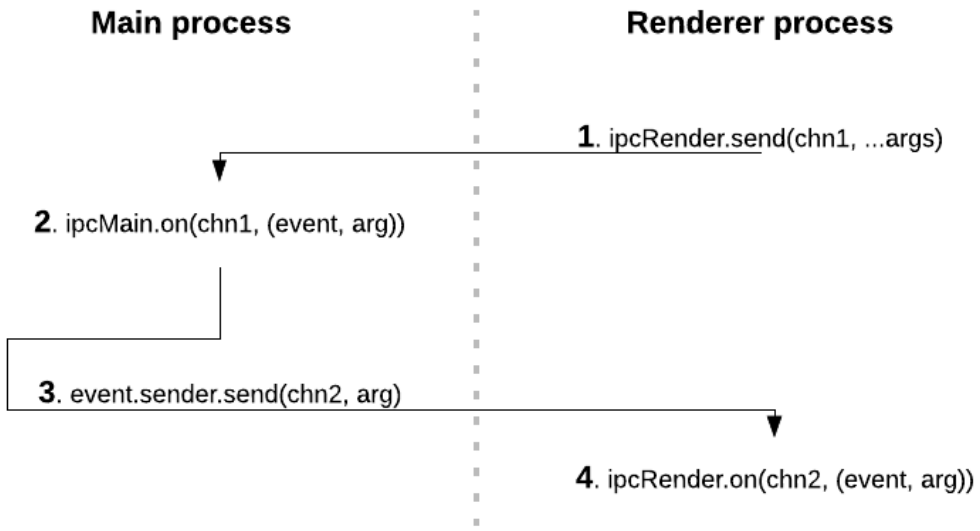


Figure 30. Asynchronous communication between processes inside Electron application

The code for the main process of the project is placed in the file **src/main/main.ts**. It initializes the application, opens the application windows and manages application lifecycle events like closing application windows or terminating an app. It is the right place for inserting code for sending data to the renderer process. Figure 31 illustrates two listeners for sending different data to the renderer. At first, SILAM output data is read and saved to the variable “data”. Next, the first listener sends a list of all available dispersed materials. The last listener sends all data about the passed dispersed material’s name. Both of them always reads data from the variable “data” to process its value and to send back desired data.

```
const data = getSilamDataFromJson();

ipcMain.on('getGasNames', (event, arg) => {
  const gasNames = getGasNames(data);
  event.sender.send('sendGasNames', gasNames);
});

ipcMain.on('getGasData', async (event, arg) => {
  const gasData = getGasData(data, arg);
  event.sender.send('sendGasData', gasData);
});
```

Figure 31. Data transfer from the main process to the renderer process (src/main/main.ts)

Now data is processed and listeners are ready for sending the desired data. The next part is to set triggers for listeners, retrieve the data and pass it to the point cloud layer.

3.8 Visualizing the processed data

When the data is accessible for the frontend, the last step is to visualize it. This section covers how the data is saved and stored. Then, the description of passing data to the point cloud layer is given. Lastly, the application window shows how the visualization is rendered and what problem was discovered about rendering 3D clouds.

Before visualizing processed data, it needs to be retrieved. The first step is to initialize variables for keeping the passed data. Figure 32 shows state variables in the React component **Main** for saving passed data and managing map interactions.
The brief description of created variables is as follows:

- **gasNames** holds a list of all available dispersed materials. It is used for selecting material from the dropdown list in the control centre.
- **selectedGas** saves a selected dispersed material. On the initial run its value is the first material's name from a **gasNames** list.
- **layer** keeps a ready point cloud layer that is applied by the deck.gl component.
- **gasData** holds all data about **selectedGas** material name. It is used for generating a point cloud layer.
- **selectedTime** saves the selected hour. It is used for viewing a cloud in a different point of time.
- **isAnimating** automatically changes **selectedTime**.

```
const [gasNames, setGasNames] = useState<GasName[] | null>(null);
const [selectedGas, setSelectedGas] = useState<string | null>(null);
const [layer, setLayer] = useState<any | null>(null);
const [gasData, setGasData] = useState<Result | undefined>(undefined);
const [selectedTime, setSelectedTime] = useState<number>(0);
const [isAnimating, setIsAnimating] = useState<boolean>(false);
```

Figure 32. State variables (src/renderer/main.tsx)

The next step is to set message senders from the renderer process to the main to trigger data processing functions. Figure 33 shows two senders: the first is asking for a list of available dispersed materials, the second function is asking for precise data about the desired material. The first sender is triggered right after application launch when the variable "gasNames" is empty. The second sender is wrapped in a function that is called automatically after retrieving the dispersed

materials' name list and on user change of selected material.

```
if (!gasNames) {
  ipcRenderer.send('getGasNames');
}

function getGasData(gas = selectedGas) {
  ipcRenderer.send('getGasData', gas);
}
```

Figure 33. Message sending functions for establishing communication between processes (src/renderer/main.tsx)

When messages are sent, the answers should be retrieved. Figure 34 shows two functions waiting for data from the main process. The first listener on the channel "sendGasNames" is waiting for a list of available dispersed materials to save them and calls another function to get information about the first material from the list. The second listener on the channel "sendGasData" is getting processed data about dispersed material to compose a point cloud layer.

```
ipcRenderer.on('sendGasNames', (event, arg: GasName[]) => {
  setGasNames(arg);
  setSelectedGas(arg[0].key);
  getGasData(arg[0].key);
});

ipcRenderer.on('sendGasData', (event, arg: Result) => {
  const newLayer = new PointCloudLayer({
    id: 'point-cloud-layer',
    data: arg.data[selectedTime].data,
    getPosition: (d: {
      position: { latitude: number; longitude: number; altitude: number };
      value: number;
    }) => [d.position.longitude, d.position.latitude, d.position.altitude],
    getColor: (d: {
      position: { latitude: number; longitude: number; altitude: number };
      value: number;
      color?: RGBAColor;
    }) => d.color || [255, 255, 255, 1],
    pointSize: 5,
  });
  setLayer(newLayer);
  setGasData(arg);
});
```

Figure 34. Message retrieving functions from the main process (src/renderer/main.tsx)

The last step is to set a control centre. The implementation in Figure 35 is very straightforward. It has a dropdown list for selecting dispersed material and a button for submitting the choice, an hour slider and a checkbox for animating a cloud.

```
<div style={{ flex: 1 }}>
          <form
            style={{ display: 'flex', justifyContent: 'space-between' }}
            onSubmit={(e: React.FormEvent<HTMLFormElement>) => {
              e.preventDefault();
              getGasData(selectedGas);
            }}
          >
            <select
              onChange={(event: React.ChangeEvent<HTMLSelectElement>) => {
                setSelectedGas(event.target.value);
              }}
            >
              {gasNames &&
                gasNames.map(gas => {
                  return (
                    <option value={gas.key} key={gas.key}>
                      {gas.name}
                    </option>
                  );
                })}
            </select>
            <input
              type="submit"
              value="Submit"
              disabled={isSubmitButtonDisabled()}
            />
          </form>
        </div>
        <div
          style={{ flex: 1, display: 'flex', justifyContent: 'space-between' }}
        >
          <input
            type="range"
            min={0}
            max={gasData && gasData.data.length - 1}
            value={selectedTime}
            onChange={(e: React.ChangeEvent<HTMLInputElement>) =>
              handleTimeChange(e)
            }
          />
          <label>
            Animate
            <input
              type="checkbox"
              onChange={
                (e: React.ChangeEvent<HTMLInputElement>) => setIsAnimating(e.target.checked)
              }
            />
          </label>
        </div>
```

Figure 35. Visualization control centre (src/renderer/main.tsx)

Finally, the application can be built to show how visualization is handled. Application execution starts with two already familiar commands “npm start dev” and “npm start”. Figure 36 shows how visualization works for 2D clouds.

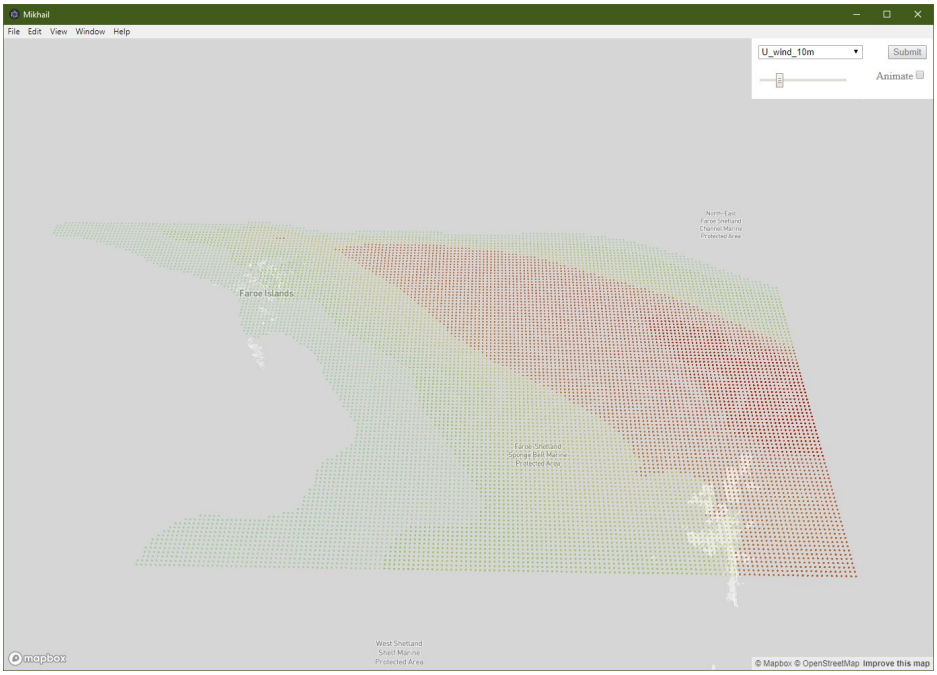


Figure 36. Point cloud of 2D cloud from SILAM data

Visualization of the 3D cloud is shown in Figure 37.

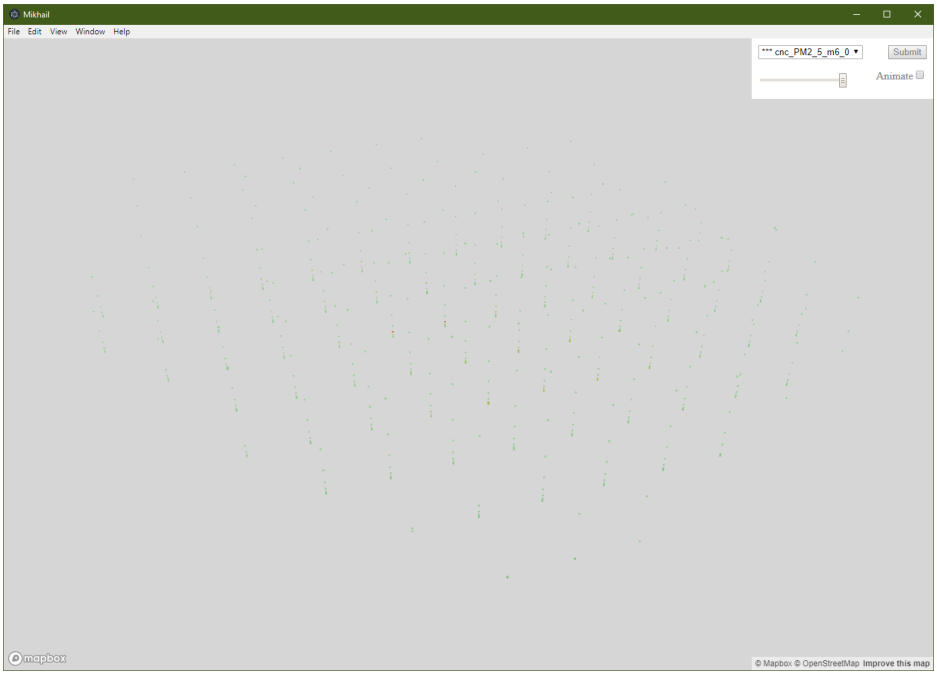


Figure 37. Point cloud of the 3D cloud from SILAM data

If the cloud in Figure 36 is somehow visible, then the cloud in Figure 37 is almost not noticeable. It reveals a problem with the SILAM coordinate grid, where the data points are physically very far away from each other. A possible workaround is to make point size bigger. However, it is not fixing the problem. Clouds in

Figure 38 looks slightly better. By trying to make the points bigger and bigger, they will start touching each other losing the cloud shape.

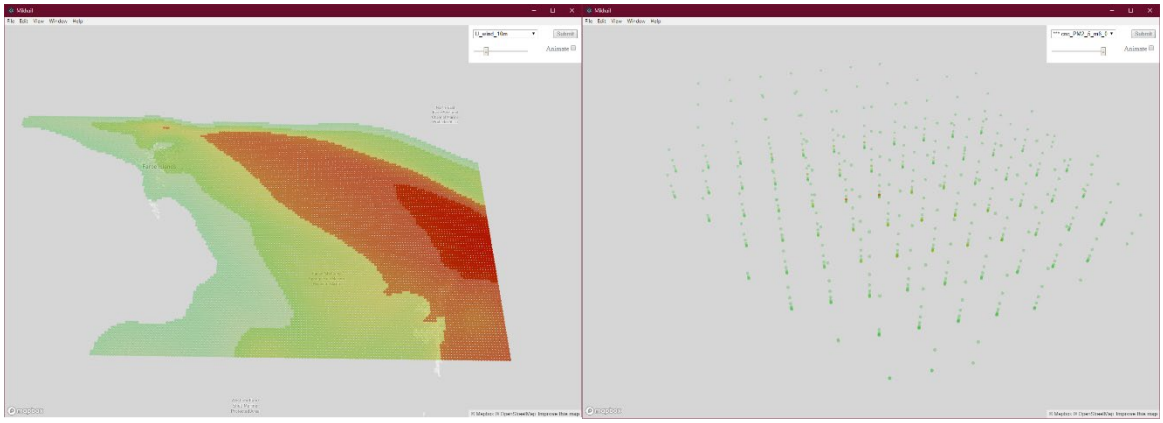


Figure 38. 2D point cloud (left) and 3D point cloud (right)

The application is fully working with all its features as it was planned. Data is read, processed and visualized. Visualization tools work perfectly, every point is a sphere that forms a true 3D figure of different colouring on the world map. The control centre does its job of changing dispersed material, hour slider shows the cloud in a different point of time and animation adds unique interactivity. Nevertheless, it was decided with the commissioner company to go further by trying to generate more data out of the available data to make 3D clouds denser. It will decrease the physical space between dots to give 3D clouds a more realistic look. Also, the enhancement of the cloud's shape will make it easier to notice.

The next sections will be a sequel to the journey of making this application. A new algorithm for generating more points between the existing points will be developed to try to make 3D clouds more noticeable and real.

3.9 Generating more data

There is no sense to lie that at first this idea of generating more data out of existing data seemed complex. There are so many data, how to collect them all correctly and what algorithm to apply for generating? These were the first questions to which could not find answers. Nevertheless, the solution to this problem was found quickly. Geographic coordinates on which points of the cloud

are lying forms a very familiar coordinate system. Coordinates longitude, latitude and altitude are perpendicular to each other what correlates to a three-dimensional coordinate system with “xyz” axes. Every data point is like points on the plot and if there is a plot, it is possible to calculate the equation to it. A plot can be built between any two points and between these two points it is possible to set more points with a graph equation. This conclusion led to the use of the interpolation technique.

3.9.1 Interpolation

Interpolation is the method of finding new values within the range of discrete values set. For example, in Figure 39 is illustrated linear graph, assume that known “y” values are only for orange dots and desired values are the green point. It is possible to calculate the value with the linear interpolation formula.

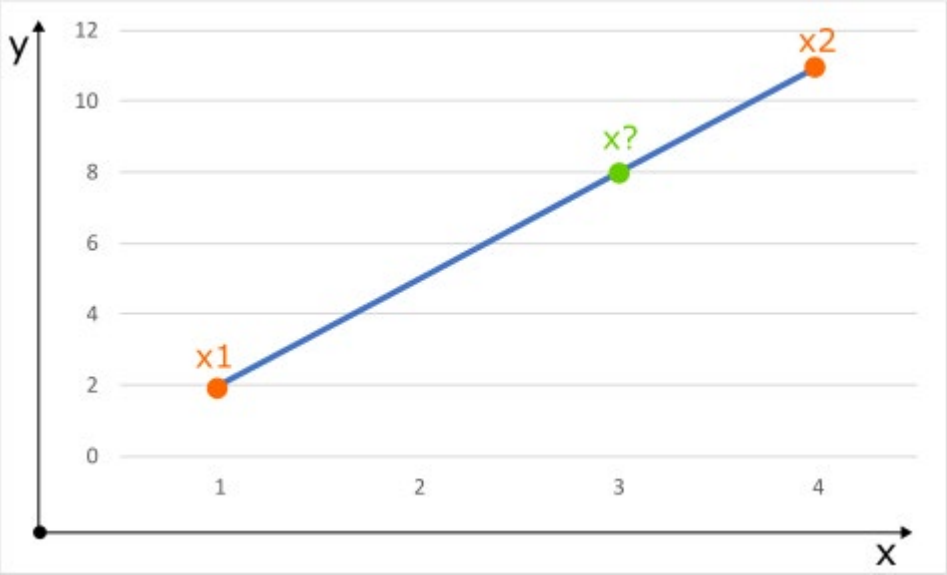


Figure 39. Graph for showing the interpolation

There are many formulas for interpolation, each is trying to minimize the error in value estimation. The easiest is **linear interpolation**, it just draws the line between two points for any value between them. More sophisticated techniques are **polynomial** and **spline interpolation** that have different formulas, but both take all available points for calculating an equation, not only two. Also, interpolation can be applied in multidimensional space, there are **bilinear** and **bicubic interpolation** for two dimensions and **trilinear interpolation** for three

dimensions. Of course, there are even more forms, but all of them are making the same thing of calculating intermediate value between known values.

As the first time working with interpolation, it was decided to pick linear interpolation. The goal of applying this technique is not to find a perfectly accurate interpolation form but to try to make 3D clouds look more natural with generated extra points. Moreover, different materials can dissolve differently, for example, nuclides due to natural subdivision might be hard to accurately estimate with only one formula.

The formula for **linear interpolation** is given on the example of unknown “y” value for the green point in Figure 39. The “y” value for the desired point can be defined with the linear interpolation Equation 1.

$$y = y_1 + (x - x_1)\frac{y_2 - y_1}{x_2 - x_1} \quad (1)$$

where	y	“y” value for unknown point	[-]
	y_1	“y” value for the first point in the range	[-]
	x	“x” value for unknown point	[-]
	x_1	“x” value for the first point in the range	[-]
	y_2	“y” value for the last point in the range	[-]
	x_2	“x” value for the last point in the range	[-]

After weeks of brainstorming the whole idea, an algorithm of calculating desired extra points was created. The whole concept is built on aligning two points on the same two coordinates when only third will be different, out of it the point value can be calculated with the linear interpolation. This approach correlates to the example in Figure 39, when for different known coordinate “x” will be different unknown value “y”.

3.9.2 Linear interpolation for a cloud of points

This subsection is describing the process of applying linear interpolation to the three-dimensional cloud of points. It involves creating a three-phased interpolation algorithm.

The interpolation will go in three different phases. For better understanding Figure 40 shows a fully interpolated cube. It displays how it would look like, if the desired number of interpolated points between known points is two. At first, let's imagine that there are no coloured points, but only black points. They represent initial data points of SILAM output data.

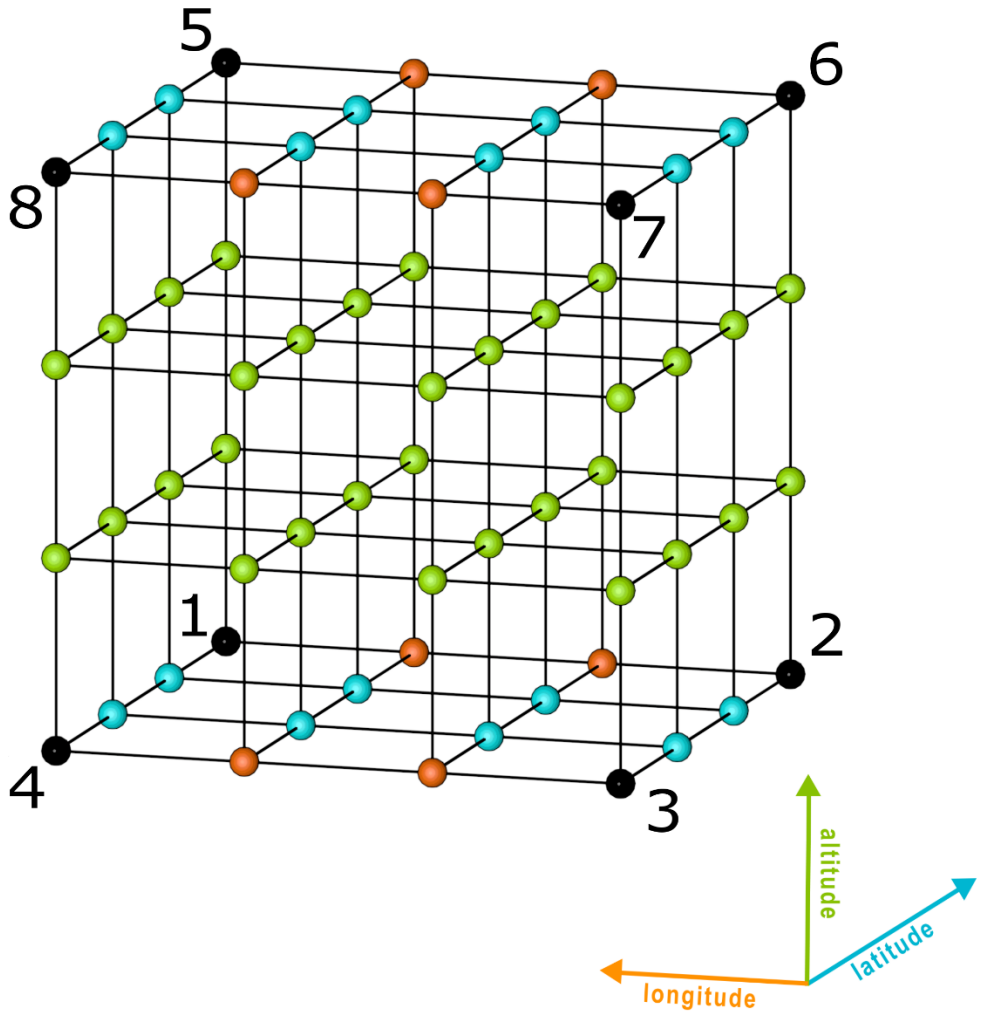


Figure 40. Graphical representation of the linear algorithm for the point cloud (Vadim Morozov, Mikhail Bobretsov)

The interpolation algorithm is done in three phases one after another, and the phases' description will refer to Figure 40:

1. **Longitude interpolation**. Every point starts looking for the closest neighbour on the left for a longitude coordinate, so that both points have same coordinates for altitude and latitude, but a different longitude coordinate. It forms pairs 5-6, 8-7, 1-2, 4-3. For every pair a graph for linear interpolation with axes "longitude – value" can be built. Figure 41 shows how longitude interpolation is working for the point pair 4-3. The same procedure goes for the rest of the point pairs, resulting in 8 orange points.

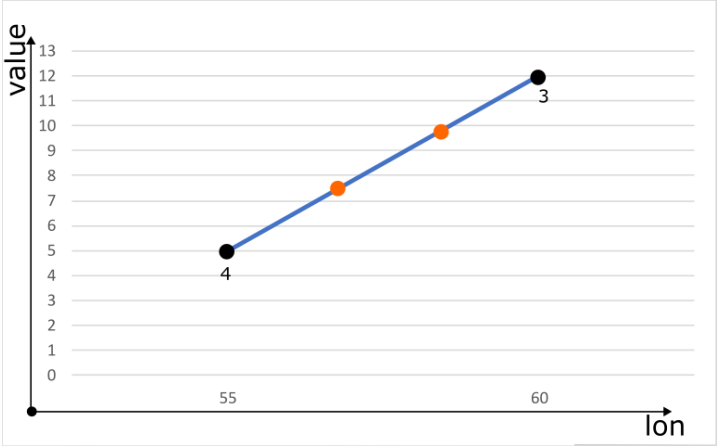


Figure 41. Longitude interpolation graph

2. **Latitude interpolation**. For the beginning, there are 16 points in total: 8 initial black coloured points and 8 interpolated orange coloured points. In this phase, every point will look for the closest neighbour point on the different latitude coordinate, but the same longitude and altitude coordinates. For black points it will form the following pairs: 8-5, 7-6, 4-1, 3-2. For every pair only the value and the latitude are different. Therefore, it is possible to create the graph "latitude – value" for linear interpolation. Figure 42 shows how latitude interpolation works for the pair 4-1. The same procedure is happening for orange points, resulting in overall interpolated 16 cyan coloured points.

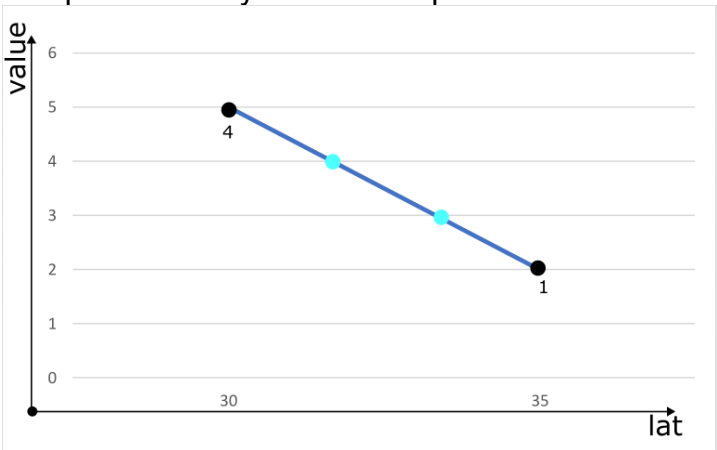


Figure 42. Latitude interpolation graph

3. **Altitude interpolation**. At first, there are 32 points in total: 8 black, 8 orange and 16 cyan coloured. In this last phase, every point will look for the closest neighbour above it on the bigger altitude coordinate, while longitude and latitude remain the same. For black points, this phase will form the following pairs: 1-5, 2-6, 3-7, 4-8. For every pair only the value and altitude are different. Therefore, it is possible to create a graph “altitude – value” for linear interpolation. Figure 43 shows how altitude interpolation works for pair 4-8. The same procedure takes place for orange and cyan points, resulting in interpolated 32 green coloured points.

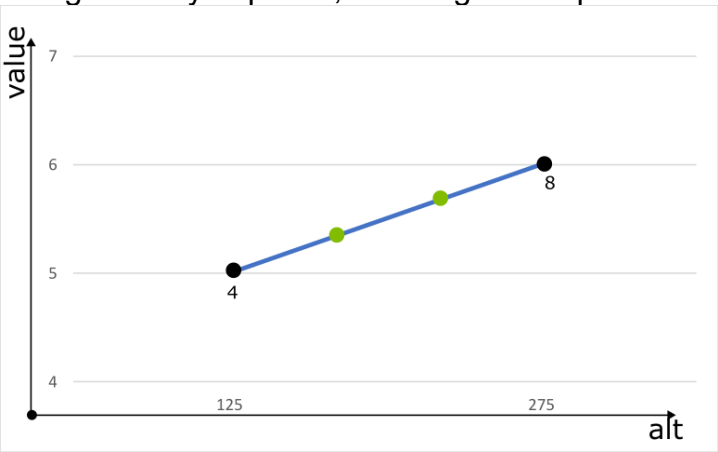


Figure 43. Altitude interpolation graph

Finally, after the described method it was managed to interpolate additional 56 points from 8 initial points with the total number of 64 points. The next step is to transfer the algorithm into code implementation and to see how it performs.

3.9.3 Interpolating SILAM data

The interpolation algorithm will be implemented inside the **src/main/utils.ts** file as a separate function. This function will take resulting data for desired material as parameter and unprocessed SILAM data under ”variables” key. In the end, it will return the same resulting data for the material but with extra interpolated points in the data points array. Figure 44 shows a piece of function that performs interpolation algorithm.

The following piece handles longitude interpolation. It implements the same algorithm described in the previous subsection. It has a variable “howManyPointsToAdd” that indicates how many points it will generate between two existing ones. Then it loops for every array of data points for all hours. Next,

it takes each point and tries to find a neighbouring point on the coordinates from the longitude array in unprocessed SILAM data. It prevents possible error, when there is no neighbouring point, but the next point is too far away, it could lead to inconsistency in the placement of generated data. Then by using Equation 1 a new point's value is calculated and the generated point is saved to a separate array. When the loop ends, all generated points are saved to start the same process for latitude interpolation, and lastly, for altitude interpolation.

```
export function lonInterpolation(
  result: Result,
  dataset: { variables: Variables },
): Result {

  let howManyPointsToAdd = 5;

  let resultData = result.data;

  let newPoints: {
    position: { latitude: number; longitude: number; altitude: number };
    value: number;
    color?: RGBAColor;
  }[] = [];

  resultData.forEach(rd => {
    if (rd.data) {
      for (let i = 0; i < rd.data.length - 1; i++) {
        let point = rd.data[i];

        let pointLonIndex = dataset.variables.lon.data.indexOf(point.position.longitude);

        if (pointLonIndex !== dataset.variables.lon.data.length - 1) {
          let nextPoint = rd.data.find(
            p =>
              p.position.altitude === point.position.altitude &&
              p.position.latitude === point.position.latitude &&
              p.position.longitude ===
                dataset.variables.lon.data[pointLonIndex + 1],
          );

          if (nextPoint) {
            let lonDiff = nextPoint.position.longitude - point.position.longitude;

            const diffToAdd = lonDiff / (howManyPointsToAdd + 1);

            for (let x = 1; x < howManyPointsToAdd + 1; x++) {
              const newLon = point.position.longitude + diffToAdd * x;
              const newValue =
                point.value +
                (newLon - point.position.longitude) *
                  ((nextPoint.value - point.value) / lonDiff);

              dataset.variables.lon.data.push(newLon);

              newPoints.push({
                value: newValue,
                position: {longitude: newLon, latitude: point.position.latitude, altitude: point.
position.altitude,
                },
                color: getColorForPoint(newValue, rd.maxValue),
              });
            }
          }
        }
      }

      if (newPoints.length > 0) {
        result.data[resultData.indexOf(rd)].data = result.data[resultData.indexOf(rd)].data!.
concat(newPoints);
      }
      newPoints = [];
```

Figure 44. Piece of the function for interpolating SILAM data points (src/main/utils.ts)

The interpolation function will be invoked in the **src/main/main.ts** file for the main process with familiar a **ipcMain** and **ipcRender** Electron function tandem. Figure 45 illustrates how interpolation function will be called.

```
ipcMain.on('getInterpolatedData', (event, arg) => {
  const interpolatedGasData = lonInterpolation(arg, data);
  event.sender.send('sendInterpolatedData', interpolatedGasData);
});
```

Figure 45. Function to transfer interpolated data to the renderer process (src/main/main.ts)

Figure 46 shows Electron functions that ask for the interpolated data and then apply the resulting data to the point cloud layer. Resulting data has the same type “Results” which frees from extra post-processing.

```
if (gasData && !isDataInterpolated) {
  ipcRenderer.send('getInterpolatedData', gasData);
}

ipcRenderer.on('sendInterpolatedData', (event, arg: Result) => {
  drawLayer(arg);
  setIsDataInterpolated(true);
});
```

Figure 46. Functions that ask for interpolation and apply resulting data to the point cloud layer (src/renderer/main.tsx)

Unfortunately, after executing the code with the interpolation algorithm it throws an error and terminates the whole application. Figure 47 shows the error code and message. There is no description for the following code without any instructions on for dealing with it. It might be an internal Electron application error referred to the application lifecycle.

```
[10344:0514/101948.759:ERROR:broker_win.cc(134)] Error sending sync broker message: The pipe is being closed. (0×E8)
npm ERR! code ELIFECYCLE
npm ERR! errno 2147483651
npm ERR! thesis-project@2.0.2 start: `electron ./dist/main.bundle.js`
npm ERR! Exit status 2147483651
npm ERR!
npm ERR! Failed at the thesis-project@2.0.2 start script.
npm ERR! This is probably not a problem with npm. There is likely additional logging output above.

npm ERR! A complete log of this run can be found in:
```

Figure 47. Error after executing the interpolation algorithm

However, code debugging revealed a bit of light on the problem. First of all, data is interpolated successfully on the main process, but the error occurs in trying to move the data from the main process to the renderer. Figure 48 shows a message about successful interpolation in the main process. The initial number of

data points is 2,300 and they were interpolated to the total number of 223,940 points. It is a lot of data, certainly more than 100 megabytes.

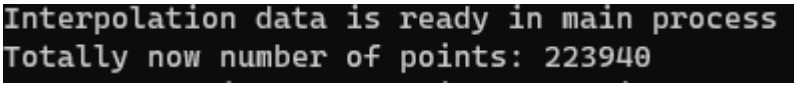


Figure 48. Command line output after interpolation

A possible problem could be that the generated data exceeded a memory buffer or after such an intense task some problem happened to the application lifecycle management. It is, however, certain that the Electron environment alone can't handle this algorithm of interpolating points. Some solutions are to run a separate REST web server that will interpolate points and send it back to the Electron application or to run a separate process in the same Node.js runtime.

However, there is still a chance that the interpolation algorithm will work in a different environment. For example, almost without changing any code, it is possible to execute this application code in a browser as a website.

3.10 Visualizing the interpolated data

This section will describe the process of migrating code from the Electron application to the React web application with subsequent visualization of the interpolated data.

A website for migrating frontend code will be built on React technology. This approach allows reducing a code rewriting to a minimum. The UI instructions and utility functions will be copied completely. The only difference is that for a web application all utility functions for processing SILAM data and interpolation will be called straightaway on the frontend side. All that is possible, because Electron shares the same technologies as modern websites do.

The React development community has created different project templates as a start for an application. For downloading a template an "npm" command will be used. Template with TypeScript and React is installed with the command "npm init react-app thesis-project --template typescript".

Figure 49 shows the project structure after executing a template script. The file **App.tsx** holds all the frontend code like **src/renderer/main.tsx**, while **index.tsx** links the React component code to the website's DOM like **src/renderer/renderer.tsx** in the Electron application.

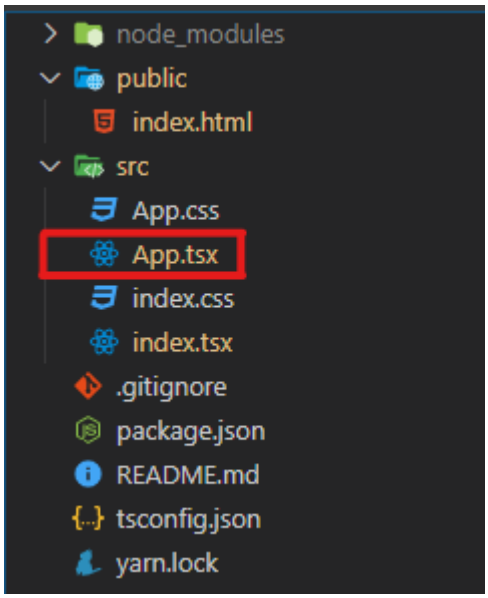


Figure 49. React web application structure

The execution of the migrated code is performed with the "npm start" command. It opens a browser window with a web application on URL "http://localhost:3000/". Fortunately, it was managed to handle such a load and visualize all the interpolated data. To make a real test, the number of interpolated points between two existing ones was set to 15. It took 2 minutes 15 seconds to interpolate all the points and show it on the screen. The total number of points was 1,530,000. The code was executed flawlessly and the result in Figure 50 was the same as expected.

The cloud looks a lot denser and real. Now it is easier to notice red areas of high concentration. Animation works smoothly drawing every second from 60,000 to 650,000 points. The performance of the visualization tools is at the highest level. All the movements around the map work without any shuttering keeping all points on the screen.

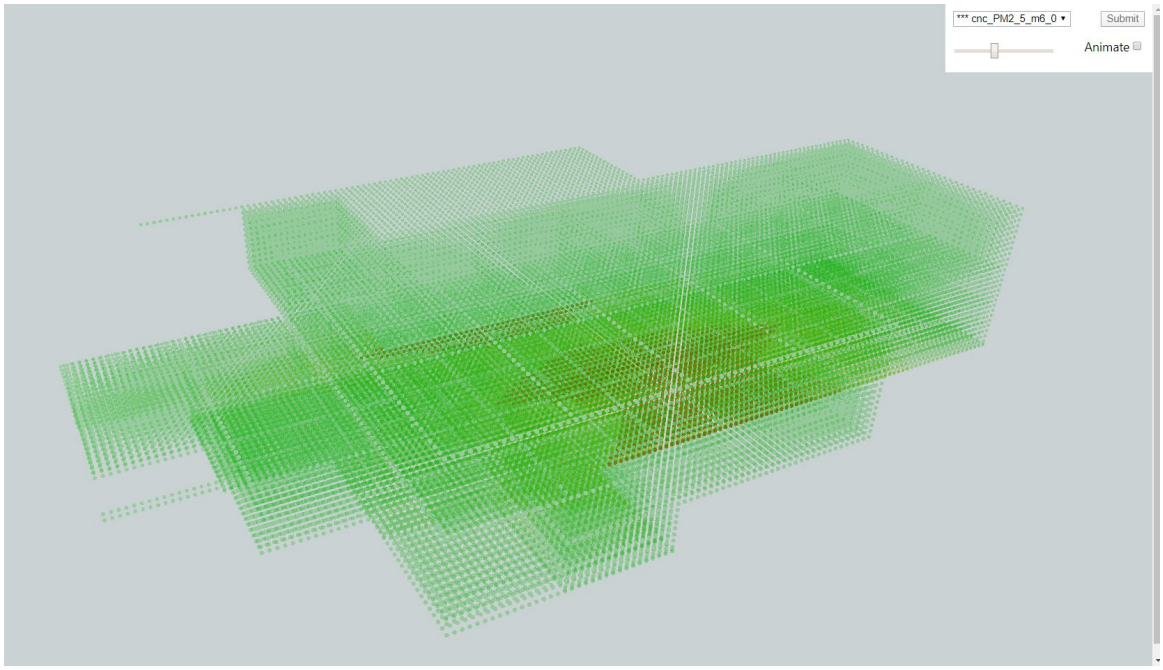


Figure 50. Successfully interpolated the cloud of points

Finally, this experiment with interpolating more points for a cloud is considered successful. The algorithm was created and implemented in code. Although it was not possible to run such a heavy task in the Electron application environment, all the code was migrated into the website environment where it worked perfectly. Visualization results fully met the expectation by enhancing 3D cloud's shape.

3.11 Evaluation of the created visualization

This section will compare the created visualization to the current solution of the commissioner company's SAS software. However, before the actual comparison, let's figure out how each solution is built. Figure 51 will help in the evaluation process. It shows the same cloud represented by both solutions.

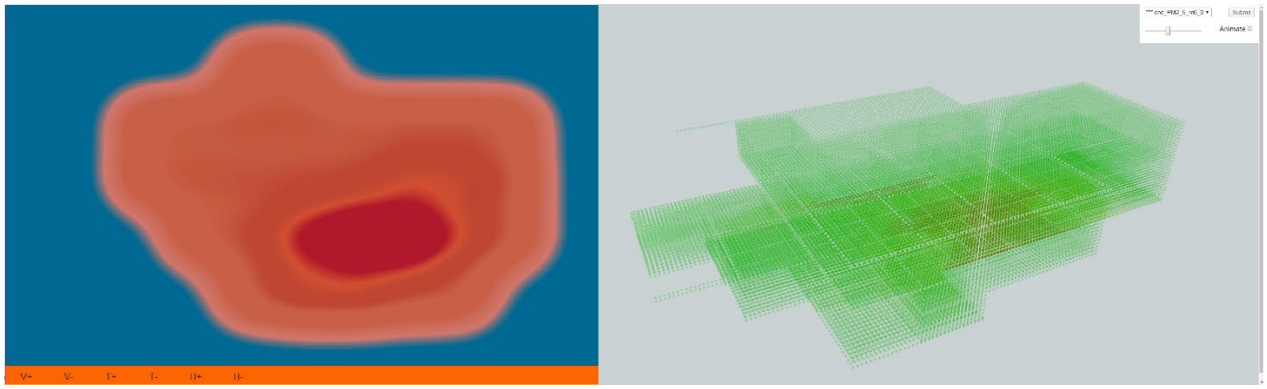


Figure 51. A comparison picture of the current solution and thesis project developed solution

The thesis solution uses thousands of rendered 3D spheres that all together form the cloud. Every sphere has a different colour and opacity based on the maximum value of the material's concentration. In this way, red areas of the high concentration are still visible at the bottom of the cloud. The control centre allows changing the selected material from a dropdown list and selecting the desired hour for the observation. Moreover, the control centre allows animating the cloud, so that, it will automatically change the cloud's shape during available hours. Furthermore, clouds rendering is performed by the deck.gl visualization library. It renders all figures in a separate canvas web element that overlays the map. In this case, all rendering is happening independently from the used map system.

The commissioner company's solution uses tools which are integrated into the map system. The visualization is performed by rendering the heatmap on top of the map. The heatmap is represented as a flat 2D layer of the coloured area. The area is divided into sectors of different colours. The colour choice is based on the maximum value of the material's concentration. The control centre is represented as a set of six buttons for switching the desired material, height layer of the cloud and observation time. The heatmap is rendered in the same canvas web element as the map system.

After understanding how each solution is built, the final step is to compare them and point out important differences. The list of developed solution's key differences is formed as follows:

- **3D clouds**. The developed solution shows the cloud as a full 3D object, while the commissioner company's solution shows the cloud as a set of 2D cloud's layers. These layers are switched by two buttons and represent the cloud state at a different height. In this, case it is much harder to estimate the whole situation because the cloud is not visible as one object but a set of switchable layers.
- **New interactions**. The developed solution has the control centre with the slider for changing an observed hour and the toggle for animating the cloud. The commissioner company's solution has only two buttons for changing the hour. In this case, the slider brings a more intuitive experience for a user. The desired hour can be set with a move of the mouse without the need to make several clicks to different buttons. Moreover, the automatic cloud animation delivers a real-life experience of

the moving cloud in the sky. This mode helps quickly see the cloud states at different time to make a prediction where it will go next.

- **Take a look from a different angle**. The 3D render of the cloud in the developed solution allows observing the cloud from different angles. A user can tilt the camera and fly over the cloud. Moreover, the user can zoom in to the specific area inside the cloud or zoom out to see it whole. While the commissioner company's solution shows only the 2D layer represented as a view on the cloud from above.
- **Separate render**. The deck.gl library renders its layers on a separate canvas web element independently from the map system. This approach adds support of GPU hardware-accelerated render method as deck.gl can use its own optimized algorithms for rendering complex 3D and 2D figures. While the commissioner company's solution renders data visualization on a single canvas web element and relies on render algorithms provided by the map system.
- **Potential for improvements**. Except for the point's colour change, the point cloud layer from deck.gl library also allows changing point's size or even adding a custom lighting. The use of these capabilities can improve the user experience by more flexible visualization. While the commissioner company's solution can only change the gradation colour of the heatmap.

The created solution has a lot of advantages compared to the previous version. Now, it can truly show the whole realistic cloud, not only the switchable layers. It is the next step of the current solution for visualizing the three-dimensional clouds. However, it can still be improved, for example, by increasing the size of points with high material's concentration. In this way, it will be much easier to notice the dangerous situation and to act properly. Nonetheless, this solution satisfies the requirements of this thesis and greatly improves cloud visualization.

4 CONCLUSION

The initial idea of the whole project was just to improve the existing solution of rendering three-dimensional clouds on the map from 2D layers to true 3D objects using the same data. Also, extra interaction capabilities were requested like showing a cloud in a different point of time and animating over available hours. Nevertheless, the described project became something more for the thesis author than just a solution to stated problems. It was a nonstop learning process with its failures and achievements.

The resulting application gathers all essential and required capabilities. The provided solution renders clouds as a set of thousands of separate dots with the different filling colour right on the map. Colour mapping implements smart technique where dots with low material concentration are more transparent to decrease distraction factor and leave highlighted what really needs to be taken into the account. Moreover, it allows observing clouds' state in different points of time with manual hour selection or automatic animation. The application brings totally unique user experience with absolute interaction: a customer can zoom in to the specific area of the cloud and zoom out to see the whole picture or even tilt the camera to estimate the situation from another angle. All of these is true with new web tools aimed at data visualization. Their capabilities allow presenting data sets with complex 2D and 3D objects together with customizable lighting, textures and shaders. Moreover, it is all available in the browser window without any prior driver or software installation.

After developing the main application new challenge was faced, three-dimensional clouds start fading and lose its shape due to the far physical distance between points. The provided solution implements the three stepped interpolation algorithm designed to generate any desired number of points. However, it was not the end, Electron environment started working unstable and killed the application for trying to pass generated data between processes. To overcome this problem, the whole codebase with its libraries was migrated to the website environment in a matter of minutes. The simplicity of this process once again proves how versatile and powerful modern web technologies are. Finally, the website with the same UI and codebase as Electron application successfully interpolated three-dimensional cloud by increasing the number of points up to 661 times. Resulting visualization shows unattainable before the result, three-dimensional clouds started looking denser and got a more realistic look. Now, it is easier to detect areas with material's high concentration, then evaluate the situation and make the right choice out of it.

During application development, thesis author opened to himself a new set of web tools for data visualization. New knowledge was gained about WebGL itself

and WebGL based libraries. Better understanding about processing complexed packed data was practised. Moreover, the unique and interesting experience was gained during developing the algorithm for point interpolation. Thesis author gets a lot of new skills and knowledge that wants to apply and improve in future work.

Finally, the thesis author and the commissioner company are sure about the success of the thesis project. However, some things can be improved and be a new start for another project. For example, it may be an installation of additional server for data interpolation to the Electron application or testing different interpolation algorithms is a good option, too. Also, a new algorithm can be developed that will be aimed to point extrapolation of clouds' edges to round corners and give clouds a more smooth look. Despite all this, provided thesis project is definitely the next step in the evolution of the previous approach and gives a fresh 3D visualization for the same data.

REFERENCES

Brigham and Women's Hospital. 2020. Slicer 4.10.2 released. WWW document. Available at https://www.slicer.org/
[Accessed 4 March 2020].

Finnish Meteorological Institute. 2014. SILAM v.5.5. WWW document. Available at http://silam.fmi.fi/index.html
[Accessed 10 March 2020].

Finnish Meteorological Institute. 2020. User-guide for SILAM chemical transport model. PDF document. Available at
http://silam.fmi.fi/doc/SILAM_v5_userGuide_general.pdf
[Accessed 10 March 2020].

Facebook. 2013. Why did we build React? WWW document. Available at
https://reactjs.org/blog/2013/06/05/why-react.html
[Accessed 24 March 2020].

Facebook. 2020a. Components and Props. WWW document. Available at
https://reactjs.org/docs/components-and-props.html
[Accessed 24 March 2020].

Facebook. 2020b. State and Lifecycle. WWW document. Available at
https://reactjs.org/docs/state-and-lifecycle.html
[Accessed 24 March 2020].

GitHub. 2020a. Electron. WWW document. Available at
https://www.electronjs.org/
[Accessed 11 March 2020].

GitHub. 2020b. Electron documentation. WWW document. Available at
https://www.electronjs.org/docs/tutorial/about

[Accessed 11 March 2020].

IKEA. 2020. Make your dream room a reality. WWW document. Available at https://www.ikea.com/ms/en_US/rooms_ideas/splashplanners_new.html
[Accessed 4 March 2020].

Jack Tomaszewski. 2018. Why TypeScript is the best way to write Front-end in 2019. WWW document. Available at https://medium.com/@jtomaszewski/why-typescript-is-the-best-way-to-write-front-end-in-2019-feb855f9b164
[Accessed 12 April 2020].

Khronos Group. 2020a. Khronos Members. WWW document. Available at https://www.khronos.org/members/list
[Accessed 2 March 2020].

Khronos Group. 2020b. WebGL Overview. WWW document. Available at https://www.khronos.org/webgl/
[Accessed 1 March 2020].

Mozilla Corporation. 2019a. Canvas tutorial. WWW document. Available at https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial
[Accessed 1 March 2020].

Mozilla Corporation. 2019b. WebGL: 2D and 3D graphics for the web. WWW document. Available at https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API
[Accessed 1 March 2020].

Medevel. 2019. 15 WebGL Medical Visualization Projects. WWW document. Available at https://medevel.com/15-webgl-medical-visualization-projects/
[Accessed 4 March 2020].

Microsoft. 2020a. Introduction. WWW document. Available at https://www.typescriptlang.org/docs/handbook/declaration-files/introduction.html [Accessed 12 April 2020].

Microsoft. 2020b. tsconfig.json. WWW document. Available at https://www.typescriptlang.org/docs/handbook/tsconfig-json.html [Accessed 12 April 2020].

Microsoft. 2020c. TypeScript. WWW document. Available at https://www.typescriptlang.org/index.html [Accessed 12 April 2020].

Mapbox. 2020a. Built with Mapbox. WWW document. Available at https://www.mapbox.com/showcase/ [Accessed 17 April 2020].

Mapbox. 2020b. Mapbox GL JS. WWW document. Available at https://docs.mapbox.com/mapbox-gl-js/api/ [Accesssed 17 April 2020].

NCO 4.9.3-alpha02 User Guide. No date. WWW document. Available at http://nco.sourceforge.net/nco.html#ncks [Accessed 10 March 2020].

Open Anatomy Project. 2017. SPL/NAC Brain Atlas. WWW document. Available at https://www.openanatomy.org/atlas-pages/atlas-spl-nac-brain.html [Accessed 4 March 2020].

Observis. 2020. ObSAS Situational Awareness. WWW documents. Available at https://observis.fi/index.php/products/obsas-software [Accessed 10 May 2020].

Uber. 2020a. AVS. WWW document. Available at https://avs.auto/#/

[Accessed 20 April 2020].

Uber. 2020b. Deck.gl. WWW document. Available at https://deck.gl/#/
[Accessed 20 April 2020].

Uber. 2020c. PointCloudLayer. WWW document. Available at
https://deck.gl/#/documentation/deckgl-api-reference/layers/point-cloud-layer
[Accessed 20 April 2020].

Uber. 2020d. React-map-gl. WWW document. Available at
http://visgl.github.io/react-map-gl/
[Accessed 20 April 2020].

Unidata.2020. Network Common Data Form (NetCDF). WWW document.
Available at https://www.unidata.ucar.edu/software/netcdf/
[Accessed 10 March 2020].