IT UNIVERSITY OF COPENHAGEN

Developing an Interactive Visualization of Bicycle Network Growth

Cecilia Laura Kolding Andersen and Morten Lynghede

May 31, 2021

Master Thesis
Software Design
Cecilia Laura Kolding Andersen, cela@itu.dk
Morten Lynghede, moly@itu.dk

May 31, 2021
Supervisor: Michael Szell, misz@itu.dk
STADS code: KISPECI1SE

Abstract

Cycling is recognized as a critical component of sustainable urban mobility. Studies show that improved bicycle infrastructure around cities can bring about environmental, health, social and equity benefits. However, even in cities considered bicycle-friendly, many disconnected components exist in the bicycle network, signifying a need to connect the whole network to encourage citizens to use bicycles for transportation. This thesis describes the process of developing a web application named BikeViz, which visualizes an algorithmically grown bicycle network connecting railway and metro stations around five cities: Copenhagen, Manchester, Munich, Paris, and Tokyo. We employ the Simple Interaction Design Lifecycle Model as the methodological approach to the development of BikeViz. The method is applied iteratively through continuous refinement of the implementation based on user feedback and clarification of user needs that occurred in the process. Furthermore, the thesis outlines the process of transforming the data to an appropriate size to ensure fast data fetching and minimal storage. In addition, we showcase the design of the user interface and the inclusion of Gestalt Principles and Jakob Nielsen's Usability Heuristics. The purpose of BikeViz is to visually demonstrate the expansion and placement of bicycle infrastructure around the given cities while simultaneously allowing for the comparison between the existing bicycle network and the algorithmically grown network. Besides the description of the development process, the final output of the thesis includes a deployed web application.

Contents

Definition of Concepts

Cascading Style Sheets

Cascading Style Sheets (CSS) describe how HTML elements are styled on a web page and used to style application user interfaces. CSS can define the color, font, text size, positioning of elements and other relevant design properties for elements and layouts[1].

Connectedness

Connectedness indicates 'the ease with which people can travel across the transportation system', and it is related to answering the question 'can I go where I want to, safely?'[4].

Directness

Directness addresses the question 'how far out of their way do users have to travel to find a facility they can or want to use?', and can be measured by how easy it is to go from one point to another in a city using bicycle infrastructure versus other mobility options, like car travel[4].

Document Object Model

The Document Object Model (DOM) is a cross-platform and language-independent API interface that interacts with any HTML or XML document. The DOM is a document model loaded in the browser. The model is represented as a tree structure wherein each node is an object representing a part of the document. Nodes can be created, moved and changed. Event listeners can be added to nodes and triggered on the occurrence of a given event[2,3].

[1] https://www.w3schools.com/html/html_css.asp
[2] https://developer.mozilla.org/en-US/docs/Glossary/DOM
[3] https://en.wikipedia.org/wiki/Document_Object_Model

GeoJSON

GeoJSON is a geospatial data interchange format based on JavaScript Object Notation (JSON). It defines several types of JSON objects and how they are combined to represent data concerned with geographic features, properties, and spatial extents[4].

JavaScript

JavaScript (JS) is a programming language used most often for dynamic client-side scripts on web pages[5].

Protected/Segregated bicycle lane

The terms "protected" and "segregated" are often used interchangeably when describing a special kind of cycling infrastructure. The terms refer to bicycle lanes physically separated from motor vehicle traffic, e.g. by curb or elevation.

Topology & TopoJSON

Topology is the study of properties of a geometric object that are preserved through deformations. TopoJSON is an extension of GeoJSON that encodes topology[6].

WebGL

The Web Graphics Library (WebGL) is a JavaScript API for rendering high-performance interactive 3D and 2D graphics within any compatible web browser without the use of plug-ins[7].

[4] https://datatracker.ietf.org/doc/html/rfc7946
[5] https://developer.mozilla.org/en-US/docs/Glossary/JavaScript
[6] https://github.com/topojson/topojson
[7] https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API

1 Introduction: Visualizing an Algorithmically Grown Bicycle Network

Cycling is recognized as a critical component of sustainable urban mobility[1, 3]. Improving the bicycle infrastructure in cities can increase commuter traffic by bicycle and bring environmental, health, social and equity benefits. On the other hand, motor vehicle dependent cities have a significant negative impact on public health, global and local ecosystems, for example by causing air pollution, and road traffic injuries. A shift to more active transport modes will have a positive impact on the environment, increase physical activity and may lead to greater social equity, as many minority and low-income communities rely on bicycles as their primary form of transportation[1].

This thesis aims to develop a browser application, named BikeViz, in React and Mapbox GL JS, visualizing the existing and algorithmically grown bicycle network, also referred to as the synthetic network, in five cities: Copenhagen, Manchester, Munich, Paris, and Tokyo. The thesis draws inspiration from previously conducted research on visualizing data and networks, combined with the experimentation of different frameworks to visualize a growing bicycle network[8]. The synthetic network is developed in an ongoing research project to grow bicycle networks in different cities[4]. The algorithm develops bicycle networks from scratch by connecting points of interest in the given cities in a step-wise process, eventually forming a fully connected bicycle network, intersecting with railway and metro stations.

BikeViz is developed to enable users to explore the synthetic network and compare it with the existing bicycle network in the beforementioned cities. The application targets two types of users, the technical user and the everyday citizen. The technical user can use the application in connection with work tasks, representing stakeholders, such as urban planners, government personnel, politicians, transport ministries, and environmental organizations. This type of user can benefit from the depiction of where to place concrete bicycle lanes in a city and how new bicycle lanes can improve the overall bicycle infrastructure[10]. The everyday citizen relates to people with an interest in the bicycle mobility options in their city and with a passion for cycling. Targeting the application at city residents can

[8] https://www.overleaf.com/read/rvmbczypvvnk

create awareness among citizens of how the bicycle infrastructure in their city can be improved, thereby increasing public support and focus political priorities[9].

BikeViz aims to provide users with a simple and intuitive platform that visualizes an algorithmically grown bicycle network and how it can be expanded efficiently by connecting points of interest around a city. The expansion is scaffolded by providing relevant statistics on improvements at each stage of the grown network and the ability to display the existing bicycle network, either on top of the grown network or by itself to allow for comparison. BikeViz enables users to study dissimilarities in the bicycle infrastructure in different city areas and assists users in learning more about the bicycle infrastructure in their city or a city of personal interest, supplying them with knowledge and data to advocate for the expansion of the existing infrastructure. Other algorithmically grown networks based on different growth strategies can be added to allow technical stakeholders to compare different bicycle development strategies to find the most suitable for their city.

This thesis describes the process of developing BikeViz, starting from the motivation behind BikeViz and the reason for developing the application in section 2. Section 3 continues with an introduction to previously conducted research to provide the basis from which the application originated. Section 4 outlines the process of eliminating data redundancy through data formatting and running performance experiments to test the new data format and displaying the bicycle network within an appropriate time frame. Section 5 introduces The Simple Interaction Design Lifecycle Model and defines user personas and the user experience in the form of use case diagrams, from which the functional and non-functional requirements emerged. Section 5 serves as the basis for section 6, Developing the Minimum Viable Product, which puts forward the class architecture of BikeViz along with valuable React and NextJS properties applied in BikeViz, concerning styling and route navigation. The section continues with implementing the functional requirements in React, NextJS and Mapbox GL JS and ends with a description of the user interface. Section 7 outlines the limitations of the development process and the application, focusing on the functional requirements that were not implemented and other objectives that were out of scope within the thesis time frame. Additionally, the section elaborates on the future steps to make BikeViz more user-oriented and to allow for the integration of more cities and additional data.

[9] https://cyclingsolutions.info/cycling-infrastructure-planning-for-the-future-of-cyclists-in-your-city/

The section concludes with a reflection on developing BikeViz by relating to the Simple Interaction Lifecycle model to evaluate the process. Lastly, section 10 concludes the whole process of implementing BikeViz and the impacts and insights it brings to cities' bicycle infrastructure.

2 Problem Definition

Even in cities considered bicycle-friendly, many disconnected components in the bicycle network exist[4], signifying a need to connect these components and the overall network. Numerous research papers and national and international guidelines are available, outlining how to strategically design and extend bicycle infrastructures by focusing on essential infrastructure design principles that encourage people to use their bicycles, such as *connectedness* (see Definition of Concepts: Connectedness), *directness* (see Definition of Concepts: Directness), and safety[2]. While these principles provide helpful and necessary guidance when planning and adding new bicycle lanes in a city, they are limited to written formats and generalizations. In contrast, the design of BikeViz attempts to visually and interactively elucidate how a bicycle network can be expanded efficiently, using an algorithm that is under development in an ongoing research project for synthetically growing bicycle networks[4]. The visualization of the synthetic network, coupled with an interactive element to expand the network, can supplement the existing research on bicycle infrastructures and provide concrete instructions to urban planners on where to place new bicycle lanes and how these benefit a given bicycle network.

Visualizing the synthetic network and related data metrics is invaluable to help users understand the significance of the data and communicate important information about how their bicycle infrastructure can be expanded efficiently. Furthermore, visualizing the bicycle network data can help users process a large amount of information at once and thereby inspire them to act on the provided information[10]. Moreover, adding new bicycle lanes on top of a city map can help users relate to the city by establishing a sense of familiarity. The integration of an interactive element, like the slider, further encourages users to explore the application and its insights. Additionally, the option to show relevant statistics of the synthetic

[10] https://www.interaction-design.org/literature/book/the-encyclopedia-of-human-computer-interaction-2nd-ed/data-visualization-for-human-perception

network helps capture the value of the network and identify relationships therein. BikeViz can be accessed via the following link: `https://bicyclevisualization.vercel.app/`

3 Initial Data Format and Selection of Frameworks

This section presents the initial data format of the synthetic network, the reasoning behind selecting React and Mapbox GL JS for development and a brief dive into the literature that provided a great source of inspiration for the user interface design. The section concludes with an introduction to the front-end technologies used in the implementation of BikeViz and the reasoning for migrating the application to NextJS.

3.1 Initial Data Format

The data initially only included Paris, with each stage of the algorithmically grown bicycle network stored in separate *GeoJSON*(see Definition of Concepts: GeoJSON) files, resulting in 40 files. Each file consisted of a GeometryCollection, with a collection of geometries containing all the coordinates for the given stage. Each element in the collection of geometries is a LineString object with a collection of two distinct coordinates, the start and endpoint of a single bicycle track, represented by latitude and longitude coordinates for each point. Additionally, the points of interest data representing railway and metro stations were contained in a GeoJSON file holding a single FeatureCollection—each feature in the collection representing a single metro or railway station and its related properties.

The App.js class[11] was responsible for importing the 40 files and storing them in an array to display the network data on the map. The hurdle of this data format was the duplication of LineString objects from previous array elements to subsequent array elements, resulting in an unnecessary amount of occupied storage.

The state of the data format at the time introduced implications for the implementation in different frameworks following: Leaflet.js and ObservableHQ. Taking the former implications into account, the final decision of frameworks and the reasoning for omitting others are presented in the following subsections.

[11] https://gitlab.com/Lynghede/bicyclevisualization/-/snippets/2126183

3.2 Selection of Frameworks

Prior to this thesis, the initial bicycle network data was partially implemented and explored in the following frameworks: Kepler.GL, Mapbox GL JS, and Mapbox GL JS paired with React. The different frameworks all come with advantages and disadvantages, some with distinct features, others very similar. A thorough examination and outline of the frameworks can be found in our research project report[12], in which we decided for our framework of choice for the bicycle network visualization to be Mapbox GL JS combined with React. The decision to select one of the different frameworks was mainly based on a matter of personal preference, along with some technical arguments: (1) Mapbox seems to be the universal open-source standard for map API's, (2) its documentation is explicit and elaborate, (3) it is easy to customize and (4) to build features on top of the map. The only implication being that both frameworks manage state and manipulate the *Document Object Model* (see Definition of concepts: Document Object Model), which may cause confusion. Alternatively, we could have chosen to do this with the native language of HTML and *JavaScript* (see Definition of Concepts: JavaScript), but this is where the personal preference of (1) React's code syntax and (2) component-based structure tips the balance of the scale.

3.3 Insights and Inspiration gained from the Research Project

This section briefly describes a selection of frameworks and articles explored in the preceding research project[12], whilst elaborating on the insights and inspiration gained from them, which has proven beneficial in this thesis.

Visually analyzing urban mobility is a research paper by Till Nagel[7] that focuses on visually analyzing urban mobility and the necessity of educating people in data literacy and which presents three data visualization projects. Data literacy has been called the next frontier in the open data movement, for the[7]:

> *"...benefit of students, employers, and society, data literacy must be recognized as a necessary civic skill."*

[12] https://www.overleaf.com/read/rvmbczypvvnk

For data to enable engagement, it must be of appropriate complexity and size, from the real world, and relatable to the viewer. Together they provide the foundation for an active learning environment, which enhances the acquisition of knowledge, skills, and competencies. The article outlines the importance of adhering visualized data to the understanding of the user in order to trigger change and exploration in regard to the current situation[7]. These perspectives are integrated in BikeViz by the depiction of an underlying map of the different cities and the option to view the existing bicycle network besides the synthetic network, adhering to the users through the familiarity with other map applications and cities.

Unfolding - a library for interactive maps in this research paper[8], the authors propose the *Unfolding* library for interactive maps and data visualization. The paper focuses on the current technological landscape of geospatial visualization tools and examines desirable design patterns for navigating and browsing these tools. The creators of Unfolding address some of the advantages and disadvantages of *geographic information system* (GIS) software and mashup tools, concluding that the software often involves a high learning curve or are intended for experts. Whereas Mashup tools are suitable for quick data exploration, only allowing a fixed set of techniques and intended for intermediate users. Unfolding's map design is built based on studies for map interactions and well-established design patterns of end-users navigation and browsing habits. The design[8] includes

- Panning the map by dragging the mouse or using the arrow keys;
- zooming by scrolling the mouse wheel or pressing + or - keys;
- double-clicking on the map centers it around a specific location and zooms in one level.

The paper provides a useful understanding of design patterns, which have proven convenient for targeting an audience seeking to explore an interactive map and adhering to common browsing habits.

Kepler.gl is a framework[13] designed to conduct geospatial data analysis. Large amounts of location data are easily visualized in the browser, and various filters and layers can be applied to the data to customize it.

Kepler.gl is a relatively simple platform to utilize and explore the visualization of data. However, when attempting to add tailored features to visualized data, Kepler.gl falls short in the customization of features and map projections as it is limited by having a predefined set of customization options. Furthermore, it is limited by the size of data elements it can process for display and it is challenging to filter data views without a time element—elements that are all crucial necessities for BikeViz. Regardless, the layout is elegant and a good source of inspiration for developing the prototype.

The research papers and frameworks presented above assisted in identifying the visualization fundamentals to emphasize when developing a web application depicting network data. This includes an understanding for the use of the application, the target audience and the functionality required to fulfil user needs.

3.4 Front-end Technologies

This subsection introduces the front-end technologies used in the development and design of BikeViz, followed by instructions on how to deploy BikeViz through Vercel.

React has become one of the most popular front-end development libraries, providing significant benefits in terms of flexible user interfaces and performance. React code has a modular structure in the form of components, written as code snippets that most often represent a part of the user interface. These components can be reused across several web pages, enabling re-usability across the application. Furthermore, it makes it easier for developers to cooperate on a project by working on individual components, thus avoiding merge conflicts[14]. In addition to the modular structure, the virtual DOM that React uses has excellent performance, as it minimizes changes to the real DOM. React observes the values of each component's states with the virtual DOM and when a component's state

[13] https://kepler.gl/

[14] https://www.peerbits.com/blog/reasons-to-choose-reactjs-for-your-web-development-project.html

changes, React compares the existing DOM object with the new virtual DOM object and finds the most effective way to update the parts of the DOM that have changed, thereby reducing unnecessary updates[15]. Another valuable feature of React is the use of Hooks, which provide components with state management and enables re-usability, as hooks are shared among components. The main Hooks used in BikeViz are useState and useEffect[16]. The useState Hook gives state to a component and an accompanying set function to update that state. Data flow and state management between components are handled by passing the set-function as props inside a component, the corresponding component, defined elsewhere, then needs to take props as its argument to fetch the useState hook. The useEffect Hook helps update components and useStates when a given value changes or after a component is rendered, as it performs side effects in function components on every re-render. A useEffect hook is used in BikeViz to fetch and set all the relevant data on the map.

These are just some of the features that make React a great library to use when designing web applications. Additionally, BikeViz is migrated to NextJS, a framework that builds on top of React. The reasoning behind this choice is outlined below.

NextJS is a React framework for production. The framework bundles the code and transforms it using a compiler, while optimizing the production build through code splitting. Furthermore, it enables static generation, server-side and client-side rendering[17]. Static generation works by generating the HTML at build time and reusing it on each request; this creates pages pre-rendered ahead of a user's request[18]. The latter becomes especially relevant for this application to reduce latency and ensure the best possible user experience as it requires fetching large data sets and quickly displaying them.

A disadvantage of introducing the framework into the application is the need for refactoring of the code. Furthermore, it is yet another framework with unique built-in functionality with which one must become acquainted to enable the use of its advantages—conclusively adding and consuming a lot of scarce hours from the project.

[15] https://www.codecademy.com/articles/react-virtual-dom
[16] https://reactjs.org/docs/hooks-intro.html
[17] https://nextjs.org/learn/basics/create-nextjs-app
[18] https://nextjs.org/docs/basic-features/pages#pre-rendering

Mapbox GL JS is a location data platform that provides custom-designed maps, usable for websites and applications. This JavaScript library uses *WebGL* (see Definition of Concepts: WebGL) to render interactive maps from vector tiles and Mapbox styles, providing developers with the opportunity to visualize large location-based data sets, with the possibility to create customized interfaces and personalized features to the map when visualizing data. The latter being one of the primary reasons for implementing the Mapbox GL JS framework.

HTML and JSX Hypertext Markup Language (HTML) is the standard markup language for web pages. HTML uses tags to define and structure the layout of a page and its different elements. A syntax extension of JavaScript, called *JSX*, is used to apply HTML in React. JSX makes it possible to write components that accept HTML quoting to be rendered in the DOM, making it easy for developers to write components.

Deployment with Vercel Vercel is a cloud platform for static sites, hybrid apps, and Serverless Functions. The deployment of BikeViz to production is tightly tied together with the Vercel platform and the stack of the platform is as follows[19]:

- Log in
- Import project from git provider
- Choose directory with the application
- Auto-detects that the application is using NextJS
- Hit deploy
- When ready, it provides a URL for the project

Furthermore, Vercel provides the developer with a neat dashboard from which the developer can access deployment information and performance analytics from different devices and logs. BikeViz can be accessed at `https://bicyclevisualization.vercel.app/`.

The combination of the above mentioned web technologies provides valuable features and in-built functionality to the implementation and design of a web application like BikeViz. The following section dives into the process of formatting the initial data format and ensuring high performance of BikeViz.

[19] https://nextjs.org/docs/deployment

4 Data Transformation and Performance

To ensure a pleasant user experience when interacting with BikeViz, the loading of the underlying map and rendering of the bicycle infrastructure data to BikeViz needs to be within an appropriate time frame. This section presents the exploration of various data formats to find the most suitable one to ensure high performance of BikeViz. The data transformation was carried out in three steps: (1) the data was reduced in size, (2) followed by performance experiments to test the retrieval of data, and (3) different approaches to displaying the new data format on the map were explored.

4.1 TopoJSON

Converting the data to *TopoJSON* was the first attempt to reduce the size of the bicycle network data. TopoJSON is generally more compact than GeoJSON and can offer a reduction of up to 80% or more, as TopoJSON files construct shapes from shared edges, instead of tracking each of them individually[20]. TopoJSON files are similar to GeoJSON files as they are stored in JavaScript objects and provide an extension of GeoJSON that encodes *topology* (see Definition of concepts: Topology & TopoJSON). The geometries in TopoJSON files are joined together from shared line and edge segments called arcs, thereby eliminating redundancy by storing related geometries in the same file.

TopoJSON was explored in an attempt to decrease the file size of each stage in the algorithmically grown bicycle network to speed up the retrieval of data and thereby browser loading. In order to do so, the original GeoJSON files were converted to TopoJSON, using the `geo2topo` command, and after that, back to GeoJSON using the `topo2geo` command. However, the size of each file increased in the final conversion, as TopoJSON is more suitable for files that contain several shared edges, rather than in this case, where it is mainly vertices that are shared between coordinates in a given file. In the final stage of Paris's algorithmically grown bicycle network, there are only nine line intersections, which account for shared vertices. The problem of duplicated coordinates is seen across the 40 files rather than in a single file. Therefore, converting each file to TopoJSON did not decrease data redundancy, as there are no duplicate LineString objects in each separate file.

[20] https://github.com/topojson/topojson

4.2 Data Wrangling

The attempt to reduce the size of each separate file did not solve the problem of duplicate LineString objects across the 40 files in the given bicycle network. Instead, to eliminate data redundancy, we developed a Python script, `datawrangler.py` attached in Appendix A, to transform the data into a shape that is more efficient for our application by converting it to a single file containing a FeatureCollection with 40 different features for each city.

First, the `datawrangler.py` script reads each file from the initial data set and sorts the files from stage 0 to 40. After sorting the files, each file's GeometryCollection and geometries of coordinates are looped over. For each LineString object, the two latitudes are combined to form a key and stored in a "global" dictionary named allDict. The allDict dictionary will eventually contain all LineString objects in the fully expanded bicycle network. Additionally, each distinct stage has a dictionary, called stageDict, to store the coordinates of the specific stage. If the key from a set of coordinates is already present in allDict, it is disregarded; else, it is stored in the given stageDict. All stageDict's are eventually appended to allDict, which is then passed to the function `write_to_geojson`, imported from the `geojsonConverter.py` script, found in Appendix B. The geojsonConverter converts each stageDict to a Feature, which is stored in the all-encompassing FeatureCollection.

The FeatureCollection consists of 40 features, each having a stage property, specifying which stage of the algorithmically grown bicycle network it represents, and a geometry, with a GeometryCollection and a list of geometries, containing all the distinct LineString objects in the given stage. The FeatureCollection is then written to a new file, used to display the different stages on the actual map.

The data wrangling provides the functionality of only having to fetch one file by saving the FeatureCollection in a state variable in the responsible class for each city. Furthermore, we see a decrease in data redundancy from *14.5 Mbit* to *8.5 Mbit* for the city of Paris, as each distinct LineString object only occurs once. See the depiction of the FeatureCollection in figure. 1.

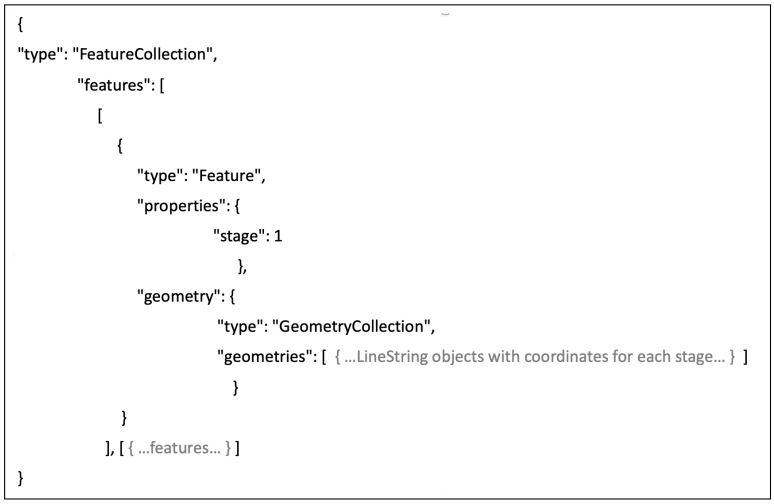


Figure 1: The data model of the FeatureCollection, containing all the stages of the algorithmically grown bicycle network.

4.3 Performance Experiments

The following section describes the performance experiments carried out on the different data formats explored when implementing the bicycle network data. Furthermore, it compares the data experiments to conclude on the best data format for the final implementation in BikeViz.

Performance Environment: The following experiments are run on the same computer. The only programs active during the experiments was either only the browser Mozilla Firefox or the browser coupled with a local server running in the terminal to ensure that other programs and processes took up as little of the computer's resources as possible, thus, making it easier to replicate similar experiments. This environment is artificial and may differ from the application's environment in everyday use.

Desktop Specifications:

- Operating system: Linux Ubuntu 20.04.2 LTS, 64-bit version

- Browser: Mozilla Firefox
- Memory: 16 GiB
- CPU: Intel® Core™ i5-3570K CPU @ 3.40GHz × 4
- GPU: NVIDIA Corporation GP106 [GeForce GTX 1060 6GB]

Initial Performance Testing: To test the performance of the initial data format, outlined in section 3.1: Initial Data Format, a function, `measureTime`, was implemented as shown below.

```
export async function measureTime(name, func, n) {
  const times = [];
  for (let i = 0; i < n; i++) {
    const t0 = performance.now();
    await func;
    const t1 = performance.now();
    times.push(t1 - t0);
  }
  const reducer = (accum, currentVal) => accum + currentVal;
  let avg = times.reduce(reducer, 0);
  avg = avg / n;
  console.log(name + " took " + avg + " milliseconds");
}
```

The function measures the time of a given function by running *n* iterations and then console logging the average of those iterations in milliseconds. The function takes three parameters: name, func, n. The *name* parameter is used for console logging the name of the function tested. The *func* parameter is the function that is reviewed and the *n* parameter is the number of iterations.

The experiment ran in production mode on a local server on the data for Paris with 40 separate files with a total size of 14.5 Mbit, giving the `measureTime` function the data fetching function and n set to 30. The experiment yielded an average of *1651.07 milliseconds*. It is worth mentioning that running the experiment with n larger than 30 resulted in the system running out of memory and the termination of the application.

To test the reduced single file data format described in section 4.2: Data Wrangling, the `measureTime` function was reused.

The experiment ran in production mode on a local server on the data for Paris with the single file approach with a size of 8.5 Mbit, giving the `measureTime` fucntion the data fetching function and n set to 100. The experiment yielded an average of *215.4 milliseconds* for fetching the data.

Conclusion on Initial Performance Testing : Comparing the performance results of the initial and reduced data format experiments, results in an improvement of:

$$1651.07ms - 215.4ms = 1435.67ms$$

Yielding a performance increase of:

$$\frac{1435.67ms}{215.4ms} * 100 = 666.51\%$$

for loading in the data.

In conclusion, removing duplicates from the data and only having to fetch one file achieves a considerable performance improvement when running in a local environment. With the current time for fetching the algorithmically grown bicycle network of Paris being *215.4 ms*. However, the deployed application needs to be tested to reach a conclusion with relevance for the performance of the final application.

Performance Testing on the Deployed Prototype: The following experiments reflect the current state of the application as of May 4th 2021. Since the last experiments, BikeViz has migrated to the Next.js framework, resulting in refactoring of the code, essentially changing the approach to fetching data. Thus, making it difficult to compare the prior experiments to the newer ones. However, the goal remains the same, decreasing the time for retrieving the data.

The experiments ran on a live server, deployed with the use of Vercel. The experiment data shows timings from all five cities and includes the algorithmically grown bicycle network, the existing bicycle network, points of interest, and statistics for each city. Thus, the measurements reflect a total of all the before mentioned data files.

The following experiments uses the `measureTime` function, with n set to 100. Furthermore, the internet connection at the given time reported a 909.36 Mbps / 68.47 Mbps download-/upload with a ping of 11 ms, which may have influenced the results.

The first experiment runs with the cache disabled and no throttling on the internet connection, the second experiment runs with the cache disabled and throttling set to regular 4G / LTE and the third experiment runs with cache.

Experiment 1: Disabled Cache		
City	Total file size in Mb	Avg. time in ms.
Paris	10.9	7.42
Copenhagen	5.2	8.84
Munich	13.2	11.88
Manchester	16.3	15.85
Tokyo	47.9	32.45
Sum	93.5	15.29 (total: 76.44)

Experiment 2: Disabled Cache & throttling regular 4G/LTE		
City	Total file size in Mb	Avg. time in ms.
Paris	10.9	18.99
Copenhagen	5.2	5.36
Munich	13.2	23.06
Manchester	16.3	28.25
Tokyo	47.9	63.65
Sum	93.5	27.86 (total: 139.31)

Experiment 3: Cache		
City	Total file size in Mb	Avg. time in ms.
Paris	10.9	12.19
Copenhagen	5.2	8.32
Munich	13.2	8.33
Manchester	16.3	17.53
Tokyo	47.9	16.87
Sum	93.5	12.65 (total: 63.24)
Second iteration: with cache built		
Paris	10.9	5.63
Copenhagen	5.2	7.66
Munich	13.2	5.27
Manchester	16.3	7.03
Tokyo	47.9	13.54
Sum	93.5	7.82 (total: 39.13)

Discussion on Performance: The fastest average speeds are observed in *Experiment 3: Cache - Second iteration: with cache built* with a total average of *7.82 ms*, which is expected as the fetching relies on the cache. The more interesting part is *Experiment 1: Disabled cache*, as it simulates a user's first visit to the application and exploration of the different cities. Here a total average of *15.29 ms* is recorded. *Experiment 2: Disabled cache & throttling regular 4G/LTE*, simulates a situation where the user is using poor bandwidth or accessing the application from a cellular device, it records the slowest total average of *27.86 ms*, which is expected as the bandwidth is decreased, thus increasing the average load time. Furthermore, it is worth mentioning that the network bandwidth may greatly vary

during the application session, resulting in significant deviations for the average time and the browser's distribution of resources, which should be considered when reviewing the experiments and the conclusion.

In order to make an attempt to compare these experiments with the other performance sections experiments, we may reduce the scope by only observing the measurements for Paris.

Comparing performance experiments for Paris			
Description	Total size	Iterations	Avg. time
40 Files	14.5Mb	30	1651.07ms
Single File	8.5Mb	30	215.4ms
NextJS(Disabled Cache)	10.9Mb	100	7.42ms

When observing the results from the performance experiments with the new single file data format, there is a decrease in an average time of:

$$215.4ms - 7.42ms = 207.98ms$$

resulting in an performance increase of:

$$\frac{207.98ms}{7.42ms} * 100 = 2802.96\%$$

Despite an increase in the total size of 2.4Mb.

When observing the total improvement from the initial to the final data format, there is a decrease in an average time of:

$$1615.07ms - 7.42ms = 1607.65ms$$

resulting in an performance increase of:

$$\frac{1607.65ms}{7.42ms} * 100 = 21666.44\%$$

Conclusion on Performance: The data fetching undergoes a significant decrease in average load time and improvement in speed-up throughout the experiments. The best performance results are yielded from the deployed NextJS version using the single file format, making up the argumentation for choosing this data format and approach going forward.

4.4 Approaches to Displaying the Reduced Data Format

Following the data formatting and performance experiments, different approaches were explored to display the data on the map in order to ensure a fast response when the user interacts with the slider. At this point, all stages of the bicycle network for a given city are stored in a single GeoJSON file containing a FeatureCollection, with 40 distinct features. The bicycle network data for each city is stored in a useState variable and updated each time the user selects a city, which makes up the foundation for each of the approaches explained in the following section.

Layer Level Property Functions in Mapbox the map functions `setLayoutProperty`, `setPaintProperty` and `setFilter` are properties at the layer level in the Mapbox Style Specification[21] that determine how layout and paint properties are rendered on a map. With this approach, the entire FeatureCollection is added to the map only once. As the slider moves, the map layer function is called upon, taking as arguments the layer's data source, the paint or layout property to be replaced, and an expression[22] that specifies the specific case and arguments to the expression. In this case, all stages starting from 0 to the slider's current value should have opacity 1 (visible), and all stages above the slider value should have opacity 0 (not visible). However, as line-opacity is a layout property and source reloads are generally necessary for changes to the layout[23], this causes the updating of line-opacity to lag, as the user interacts with the slider. Additionally, expression validation is generally considered slow, which might also cause performance issues[24]. The setPaintProperty() function is presented below as an example of a layer level function.

[21] https://docs.mapbox.com/mapbox-gl-js/style-spec/layers/
[22] https://docs.mapbox.com/mapbox-gl-js/style-spec/expressions/
[23] https://github.com/mapbox/mapbox-gl-js/issues/7459
[24] https://www.gitmemory.com/issue/mapbox/mapbox-gl-js/7235/491700096

```
useEffect(() => {
    if (!isMapboxLoaded) return;
    const source = mapInstanceRef.current;
    if (!source) return;
    if (source.getLayer("paris_paths")) {
      source.setPaintProperty("paris_paths", "line-opacity", [
        "case",
        [">", ["get", "stage"], slider],
        0.0,
        1,
      ]);
    }
  }, [slider]);
```

Array Slicing is an in built JavaScript function that returns a desired part of an array from element x to y exclusive. The array slicing approach uses a custom useEffect hook to interactively update the bicycle data according to the slider's current value by slicing the entire FeatureCollection to show the desired stages of the network. The hook first examines if the underlying map is loaded; if not, the useEffect stops. Secondly, it checks whether the map's source layer that shall hold the bicycle network data has been added to the loaded map. If this is the case, the useEffect creates a new dictionary variable to store the data desired for display, mimicking the original FeatureCollection. The dictionary updates with the relevant data from the useState variable that stores the original FeatureCollection, after slicing the FeatureCollection from index 0 to the slider value + 1. The newly built dictionary now represents the data of the map's bicycle-path source layer. Whenever the map is loaded or the slider value changes, the useEffect runs, ensuring that it does not run at unnecessary times, as this would cause the application to slow down.

The impact of the visualization on the map is that it draws the LineStrings objects on the map with relative speed, displaying the correct data within a fraction of a second. However, it does not achieve a smooth drawing of all LineString objects, which is demonstrated by observing that some lines are drawn ahead of others. The array slicing approach is presented below.

```
// Setting the "bicycle_path" data of slider
  useEffect(() => {
    // Check if the map is loaded
    if (!isMapboxLoaded) return;

    const source = mapInstanceRef.current.getSource("bicyclepaths");
    // Check if the source layer is added to the current map
    if (!source) return;

    // Temporary dictionary variable to store the desired data
    const sliderData = {
      type: "FeatureCollection",
      features: [],
    };

    // Updates the dictionary with the desired data
    newData.path.features
      .slice(0, slider + 1)
      .forEach((element) => sliderData.features.push(element[0]));

    // Update the map with the data
    source.setData(sliderData);

    // Makes sure the useEffect is only run when the map is loaded or slider value
        changes
  }, [isMapboxLoaded, slider, router.query.id]);
```

Based on the performance experiments and the different approaches to display the network data, the reduced single file data format, combined with the array slicing approach was chosen for the final integration of the data in BikeViz.

5 Methodology: A Simple Interaction Design Lifecycle Model

The following sections present the methodological approach behind the development of BikeViz. The first section presents the Simple Interaction Design Lifecycle Model and a brief description of the essential characteristics of the model. Followed by an identification of the users and the development of user scenarios and use case diagrams, which lay the foundation for establishing functional and non-functional requirements. In addition, the functional requirements serve as the basis for the two Figma prototypes presented at the end of the section.

5.1 The Simple Interaction Design Lifecycle Model

The Simple Interaction Design Lifecycle Model[9] lays the foundation for the process of developing BikeViz. The model emphasizes a user-centered approach, where users are involved in design processes throughout the development to ascertain that the product adheres to the users and their expectations. When working with a user-centered approach, three crucial principles need to be captured: (1) defining the user and their interaction with the system at hand, (2) empirical measurements, thereby facilitating user interaction with the product, in the form of testing prototypes and usability testing, and (3) iterative design, by continuously testing the product on users and refining based on user feedback.

The four basic activities of interaction design include:

1. **Establishing requirements**
   During the first phase of the lifecycle model, it is essential to focus on the users and to clarify their specific needs, which feeds into the process of establishing functional and non-functional requirements for the Minimum Viable Product and prioritizing them according to their importance to the user.

2. **Designing alternatives**
   In the second phase, design alternatives are developed and presented to the user to gain their perspective on the variety of designs and evaluate product design based on their needs.

3. **Prototyping**
   The third phase concerns the development of prototypes with similar functionality to the final product, allowing the user to interact more in-depth with the product and enable more elaborate testing of the interaction.

4. **Evaluating**
   Lastly, the product is evaluated based on functionality, usability and acceptability of the product and design, according to the user experience.

The model applies an open and straightforward approach to the design of an interactive product. The abovementioned activities are related to each other and therefore used in an iterative and overlapping matter to return to previous activities based on user feedback, proving helpful to developers to ensure that the application fits users' needs. The Simple Interaction Design Lifecycle model is displayed in figure 2 below.

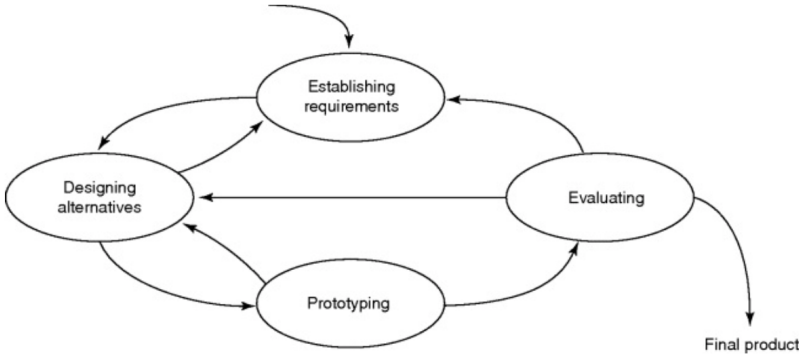


Figure 2: The Simple Interaction Design Lifecycle Model[9]

5.2 The User Experience

The Simple Interaction Design Lifecycle model's first activity concerns the identification of users and user scenarios, to establish the requirements of the application. User experience involves designing products around people rather than teaching people how to use products. In order to design around people we must understand them - their behaviors, attitudes, needs, and goals. To achieve the enlightenment of such qualities, a persona is

employed to encourage design decisions based on a natural person's needs and not on a generic and undefined user[25].

5.2.1 User Personas

A persona is a fictional yet realistic description of a typical or target user of the product and the description of personas must reflect those of real people. The description of a user persona should include background information, job descriptions, concerns, and goals to help developers gain a common understanding and to refer back to a standard and agreed upon user when designing a user-oriented product[25].

The thesis defines two user personas. The first persona, Elisabeth Jensen, uses BikeViz in connection with her work tasks and is regarded as the technical user. The second persona, Marcello Rodriguez, is regarded as the everyday user and utilizes the application out of an interest in the bicycle infrastructure in his current city of residence and to explore the existing bicycle network. The two personas are presented in figure 3 and 4.

[25] https://www.nngroup.com/articles/persona/

DEMOGRAPHICS

Name	Elisabeth Jensen
Nationality	Danish
Age	36
Family	Married
Occupation	Urban Planner @ The Technical and Environmental Administration of Copenhagen
About	Mother of two, environmentally aware, a frequent user of her bicycle when moving around the city.

JOB DESCRIPTION

Elisabeth is in charge of urban planning and overseeing the expansion of the existing bicycle infrastructure. Furthermore, ensuring safe bicycle mobility opportunities in Copenhagen with a focus on environmental sustainability.

PRODUCT USE

When deciding where to place new bicycle lanes, focusing on cost-efficiency and connectivity, and comparing the current and synthetic bicycle network to visually gain an idea of where new bicycle routes may best connect the city. Furthermore, to obtain relevant statistics regarding the expansion of the bicycle infrastructure.

GOALS

- To increase transport by bicycle and simultaneously decrease transportation by car or other non-CO2 friendly transport alternatives.
- To ensure significant gains in bicycle mobility around Copenhagen and secure investments.
- Adhere to local, national and international climate interventions and goals regarding sustainable transportation.
- To ensure safe travels for all bicyclist.

CONCERNS

- To stay within budget.
- To ensure outside investment and support for work-related projects.
- The comprehensiveness of the various aspects that goes into expanding the bicycle infrastructure.
- Meeting the external and internal requirements from stakeholders.

Figure 3: User Persona 1: Elisabeth Jensen

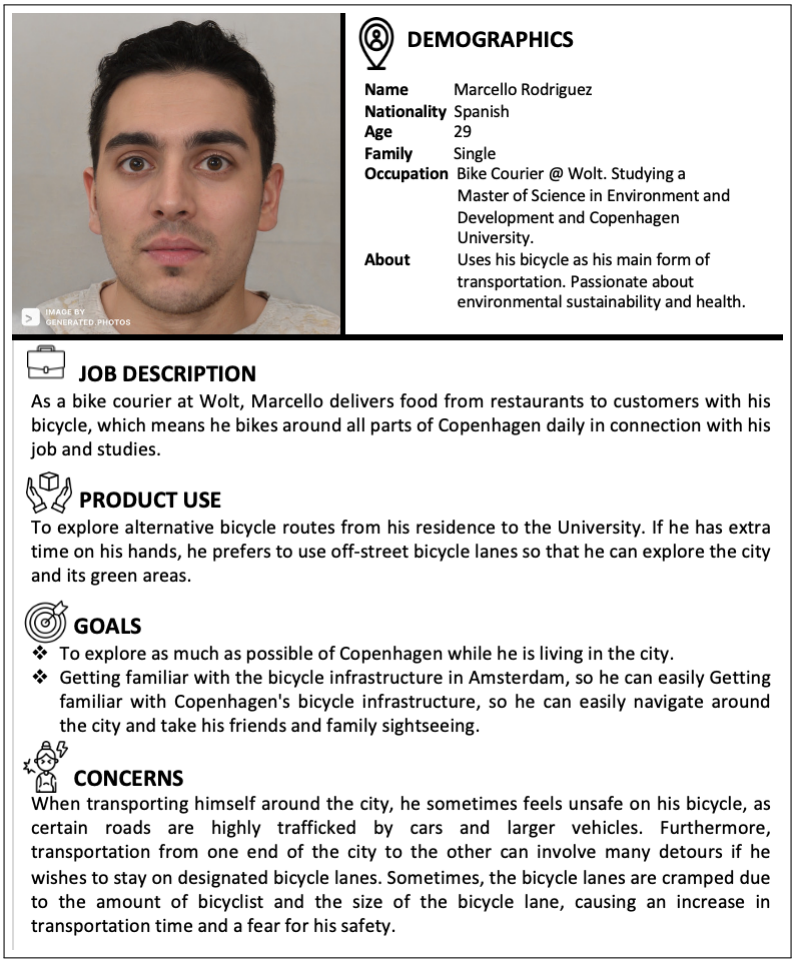


Figure 4: User Persona 2: Marcello Rodriguez

The purpose of designing two distinct personas is to target BikeViz at users who utilize it for technical purposes and traditional city inhabitants utilizing it out of interest. The technical user may benefit from the insights it brings about the current bicycle infrastructure in various cities and its expansion possibilities. On the other hand, the everyday citizen may find it interesting to explore and learn more about their city's bicycle infrastructure.

5.3 User Scenarios and Use Case Diagrams

User scenarios are developed to explore how users interact with BikeViz and to determine the main functionality and user interface requirements. In contrast, use cases are an abstraction of an interaction with the target system that determines a delimited use of part of the system[6]. The development of user scenarios and use case diagrams are helpful to determine how users interact with BikeViz and are tightly coupled with the user personas developed to describe the application's users, as presented in the previous section. Presented below are two user scenarios for each user, along with corresponding use case diagrams.

User Scenario 1: Elisabeth Jensen There is an excess in the budget for urban planning, which has been allocated to the expansion of the bicycle infrastructure in Copenhagen. The user wants to find out where it would be most beneficial to invest the allocated budget to efficiently expand the current bicycle network, with the least number of invested kilometers, to stay within the budget.

User Scenario 2: Elisabeth Jensen Following Scenario 1, the user needs to pitch the expansion of the bicycle infrastructure in Copenhagen to stakeholders. In the pitch, she wants to present them with factual data metrics highlighting the benefits of investing in more dedicated bicycle lanes to optimize the existing bicycle infrastructure.

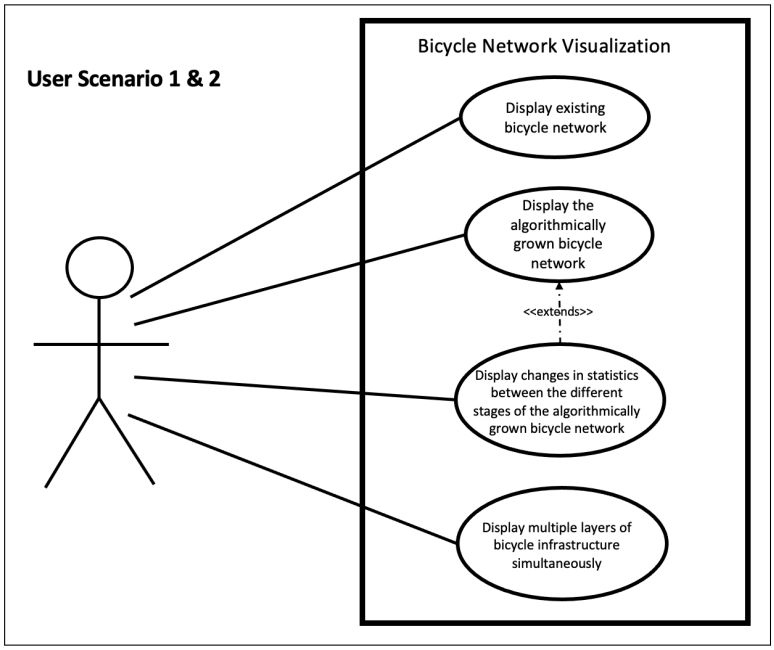


Figure 5: User Scenario 1 & 2: Elisabeth Jensen. The stick figure represents the actor, the oval shapes represent the use cases, and a straight line represents communication between the actor and use cases. Additionally, the «extends» element specifies a use case relationship, meaning the extending use case is optional and dependent on the extended use case.

User Scenario 1: Marcello Rodriguez The user wants to explore the city from a new perspective in his spare time, thus varying from the usual everyday commute to university. He decides to plan a bicycle route that mainly uses off-street bike lanes to avoid constant car traffic noise and stress.

User Scenario 2: Marcello Rodriguez Friends from abroad visit the user and give him the task of showcasing the city. The group chooses to rent bicycles to be able to cover more ground quickly. The user wants to plan a bicycle route covering all of the city's museums (points of interest). The route should only rely on segregated bicycle lanes as the user wants to accommodate his inexperienced cyclist friends. Finally, the user wants to share the route with the group, so the planned route is accessible to everyone.

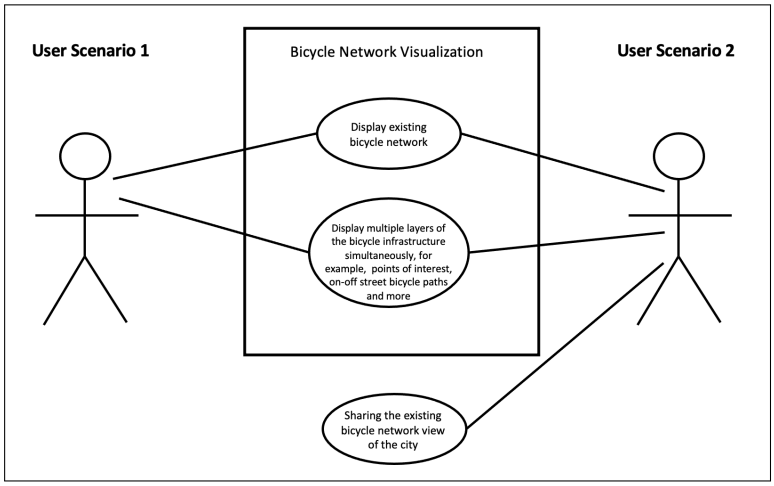


Figure 6: User Scenario 1 & 2: Marcello Rodriguez. The stick figure represents the actor, the oval shapes represent the use cases, and a straight line represents communication between the actor and use cases. The use case positioned outside of the frame is not intended for implementation but rather a possible future feature of BikeViz.

The two user personas, coupled with proposals made by our supervisor, lays the foundation for identifying the functional requirements in the following section. In the remainder of the thesis, the reference to a specific user concerns our supervisor, acting as a user and stakeholder to BikeViz.

5.4 Functional and Non-Functional Requirements

The following section is divided into three parts: the functional requirements specifying the functionality of BikeViz, followed by the prioritization for the implementation of these functional requirements, and the non-functional requirements posing constraints on the application's development and functionality.

The Functional Requirements are identified and prioritized according to their importance to the user.

1. As a user, I want to be able to access information regarding the purpose and usage of BikeViz, because I want to know how I can utilize it.

2. As a user, I want to be able to select my city or a city with personal relevance to me because I want to explore the bicycle network in the given city.
3. As a user, I want to be able to choose between the existing and the synthetic bicycle network because I want to compare the two.
4. As a user, I want to be able to customize the visualization of the map design because I want to discover the bicycle networks in different layouts.
5. As a user, I want to see statistics regarding the altered data metrics of the synthetic bicycle network because I want to see the progression at each stage.
6. As a user, I want the freedom to be able to interactively navigate between the different stages of the synthetic bicycle network to explore how it expands.
7. As a user, I want to be able to access documentation on how to navigate BikeViz, to gain in-depth knowledge on how to utilize it.
8. As a user, I want to be able to see the core functionality of BikeViz when I enter the website, to engage with the application immediately.
9. As a user, I would like a step-by-step guide on how to navigate the application, in case I am uncertain of how to perform certain tasks.
10. As a user, I would like to compare two different cities simultaneously because I want to compare their existing or synthetic bicycle network.
11. As a user, I want to be able to share the bicycle network view I am currently exploring with others because I want to convey the information I have obtained to people of interest.

The MoSCoW prioritization technique is used to prioritize functional requirements according to:

- M: Must-have initiatives: are essential for the prototype's success, and their absence should be considered a failure of the prototype.
- S: Should-have initiatives: are important but not necessary for the final prototype.

- C: Could-have initiatives: are desirable initiatives to improve the user experience, but not essential.
- W: Won't-have initiatives: are the least-critical and are not accounted for in the prototype's implementation.

Functional Requirement	MoSCoW prioritization
1	M
2	M
3	M
4	M
5	M
6	M
7	M
8	S
9	S
10	C
11	C

The Non-functional Requirements specifies how to implement certain functionality in BikeViz and generally poses as limits on its development and functionality.

- A time constraint of 4 months to complete BikeViz.
- Response time for loading the algorithmically grown bicycle network onto the underlying map should take no more than 10 seconds, preferably less than 5 seconds[26].

[26] https://medium.com/@slhenty/ui-response-times-acec744f3157

- A loading screen to keep users occupied while awaiting the fetching of data sets.
- Storing the bicycle network data for different cities locally and fetching the data within an appropriate time.
- The application's graphical user interface should be intuitive for the user to navigate.
- Different toggle options for the user to choose between different map filters, layers and cities.
- A slider that iteratively shows the different growth stages of the synthetic bicycle network.
- A statistics window for displaying relevant data metrics of the different synthetic network's stages.
- The system must meet Web Content Accessibility Guidelines WCAG [27].

The requirements presented above guides the implementation of BikeViz, through the identification of must-have initiatives and system requirements that are considered essential for the functionality of the Minimum Viable Product.

5.5 Figma Prototypes

The second activity of The Simple Interaction Design Lifecycle model is concerned with the development of design alternatives, resulting in the design of two Figma prototypes.

Figma is a cloud-based design tool that follows a component-based structure, allowing the extraction of *Cascading Style Sheets* (see Definition of Concepts: Cascading Style Sheets), proving valuable when migrating the Figma designs to the Minimum Viable Product. The development of design alternatives in Figma and, eventually, the Minimum Viable Product are based on the *Gestalt Principles*, presented in Appendix C and Jakob Nielsen's *Usability Heuristics* in Appendix D.

The prototypes illustrate a use case, where the user is first presented with a tutorial navigating the user through the application's features, which eventually redirects to the application's landing page. The landing page presents the user with an underlying map depicting

[27] https://www.w3.org/TR/WCAG21/

Paris and a sidebar with the options to choose between cities, network views and map filters. However, due to the nearly impossible task of implementing map functionality in Figma, the map only displays the synthetic bicycle network or a plain city view of Paris. Furthermore, as the primary purpose of the Figma prototypes is to demonstrate BikeViz's visual design, only limited functionality is implemented.

The primary differences between prototype v1 and v2 are the sidebar's design, the choice and addition of icons, and the altered positioning of different calls to action. In prototype v1, the sidebar elements display the different cities a user can choose from, along with a documentation icon leading to a documentation page. In prototype v2, the sidebar component contains three different tabs, one to choose between cities, one to choose layers and one for filters, to group the functionality that causes changes to the map in the same component. Additionally, the documentation icon is positioned at the top right of the page, along with a few other icons, including statistics, tutorial, sharing, and information icons, to gather functionality that might be more relevant to the application's technical users. The tutorial icon allows the user to start the tutorial at any point in time. As the tutorial is running, the user has the option of skipping the tutorial at each step. Alternatively, the statistics window from prototype v1 is not implemented in v2, as the styling of the window is still uncertain at this point.

Prototype v1 has been tested on our user and can be found here: `https://www.figma.com/file/aNZXibXT90ex5sUyjS51hF/Prototype-v1-25-02-2021`.

Prototype v2 is the revised result of the consultation with our user and can be accessed here: `https://www.figma.com/file/xnlnQpxgmV1ZQPkzh5zub7/Prototype-v2-08-03-2021?node-id=0%3A1`. To explore and interact with the Figma prototypes, press the play icon in the top right corner of the screen.

The final user interface of BikeViz is based on prototype v2, with several adjustments and additions based on feedback from our user. The following section corresponds to the third activity of The Simple Interaction Design Lifecycle Model and describes the implementation of the functional requirements in React, NextJS and Mapbox GL JS.

6 Developing the Minimum Viable Product

The following section describes the class architecture of BikeViz and the implementation of the functional requirements ranked as must-have initiatives according to their importance to the user. The segment is followed by the inclusion of Jakob Nielsen's Usability Heuristics in the user interface components and the Gestalt Principles in BikeViz's user interface design. As most non-functional requirements are mainly concerned with usability perspectives, their implementation is integrated into the functional requirements specification.

6.1 Class Architecture

Based on the structure of prototype v2 presented in section 5.5: Figma Prototypes and the decision to build a browser application using NextJS, the class architecture in figure 7, was applied for the code. For a larger depiction of the class diagram, navigate to Appendix E.

The pages folder contains every file that is either a route or an endpoint for the web application. The `_document.js` class is in charge of defining the layout for BikeViz and injecting custom styles, while the `_app.js` class initializes the pages of BikeViz, where each city has a page. The `index.js` class forwards the user to the landing page when entering the application, which depicts the city of Paris. The pages folder is further divided into two subfolders.

- **The city folder** consists of the `[id].js` file that creates a dynamic route for each city, passing the id and the data of the user-selected city. Furthermore, the class defines how to assemble the user interface by calling the different components and passing properties and state logic down the component tree.
- **The documentation folder** contains another `[id].js` class, in charge of initializing the various pages that make up the documentation page.

The components folder contains the components that are persistent across all pages, including styled-components specific to each component. The sidebar component is a parent to the layers, filters and location components and the navigation bar component a parent component to the information and statistics component. In comparison, the documentation and loading screen files are stand-alone components. The loading screen is displayed

when the user enters BikeViz to keep the user's attention while fetching the data and the documentation component is called from the navigation bar.

The lib folder contains the files in charge of fetching and passing on the bicycle network data and documentation pages. The files ending with *Fs* fetches the data at build time and builds the different dynamic routes. In contrast, the remaining files in the folder are in charge of redirecting the user and fetching new data for the different routes when the user selects a new city.

The map folder consists of the `mapFunctions.js` class, which contains the functions concerned with displaying the various bicycle network data layers on the map.

The public folder contains the `CityData.js` class, which implements a dictionary with the different cities, countries, city center coordinates and zoom level. The dictionary provides the map with a location to fly to and a zoom level for the given city. The public folder contains the subfolder *Cities* which consists of a folder for each city, responsible for storing the algorithmically grown bicycle network, statistics of the grown network, points of interest and the existing network.

The documentation folder holds the contents of the various documentation pages.

The styles folder holds the files for specifying the global CSS of the application.

The utilities folder contains the files for wrangling the bicycle network data and measuring performance.

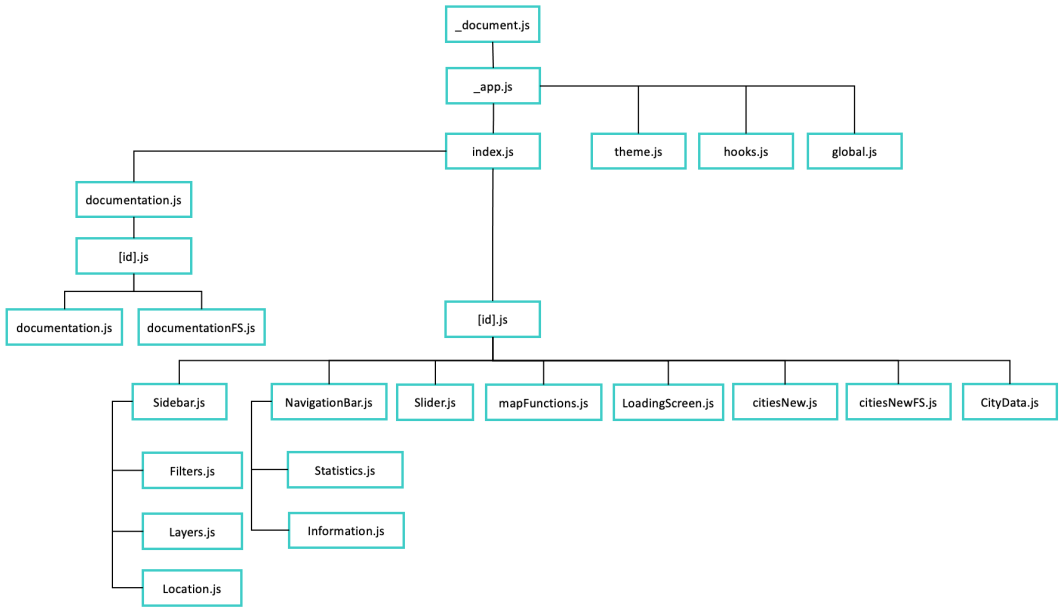


Figure 7: The Class Diagram: Architecture of the JavaScript files in BikeViz

Pages and Routes The main page of BikeViz is a dynamic route called *City* which initializes with the city of Paris. The City page uses dynamic routing to make custom URLs for each city in the data set by fetching the entire data set and mapping each city to its URL with the prefix "city", followed by that specific city, resulting in:

https://bicyclevisualization.vercel.app/city/[name of city]

The dynamic route's purpose is to initialize the map with everything that comes with it, including the map sources and layers. The sources are where the data is set, which is dependent on the name of the city and the layers are responsible for the styling of the map and the visualized bicycle networks.

The dynamic route is also in charge of the initial rendering of BikeViz's remaining components. The reasoning is that the map constitutes the primary functionality of the application. The remaining components of the application either alter or are dependent on the data the map instance provides, thus placing the map in the top of the class hierarchy as a parent component and the rest of the components below as child components, enables the passing of relevant data from parent to child.

Styled-components and Themes Each class in the components folder in BikeViz has its corresponding styled-components integrated within that class, which specifies UI elements unique to the given component. Instead of using a CSS file to style different HTML components, creating components is done through the library *Styled-Components*[28]. The styled-components are used inside the various components classes and at times in separate JavaScript files, for components reused across the application by declaring and exporting an arbitrary number of components styled with CSS properties into the relevant components. Styled-components then keep track of which components to render, resulting in less unused loaded code and eliminating the need for inventing different class names and the correlating bugs that comes with it. Furthermore, it makes deletion of dead CSS-code much easier, as tooling can detect unused components.

In addition, the application uses a helper function to generate a unique styled component that handles global styles, namely `createGlobalStyle`. Usually, styled-components are automatically scoped to a local CSS class and therefore isolated from other components. The global style sheet removes this limitation, and things like CSS resets, or base stylesheets can be applied[29].

Finally, the application uses a theme file. The file consists of two style themes: light and dark. The theme is referenced across the entire application in multiple styled-components, to enable dynamic styling that changes corresponding to the active theme selected by the user.

The class diagram modelling guided the code implementation in NextJS and was adjusted iteratively through the development process to accommodate new features and user requirements. The following section elaborates on the individual components in relation to the functional requirements.

6.2 Implementing Functional Requirements

The following section describes the implementation of the functional requirements ranked as must-have initiatives according to their importance to the users.

[28] https://styled-components.com

[29] https://styled-components.com/docs/api#helpers

Functional Requirement 1 *As a user, I want to be able to access information regarding the purpose and usage of BikeViz, because I want to know how I can utilize it.*

When a user enters BikeViz, after the loading screen has faded, a welcoming message immediately greets them, describing the purpose and usage of the application. The message is a pop-up located on the right side of the screen, anchored to an information icon, signalling that the user can access information here, as part of a navigation menu that is always visible throughout the application. Clicking on the icon makes a border appear around it and opens the information window. The purpose of the information window is to scaffold the user's experience by briefly introducing BikeViz's purpose and core functionality. The information message can be seen in figure 8.

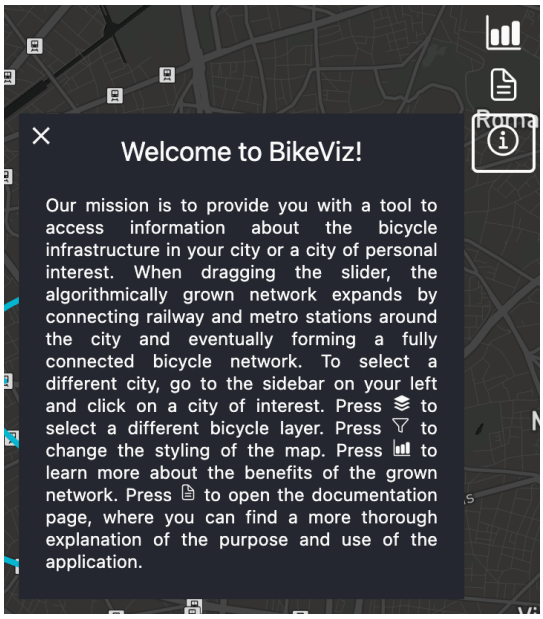


Figure 8: The Navigation Bar and Information Window

Functional Requirement 2 *As a user, I want to be able to select my city or a city with personal relevance to me because I want to explore the bicycle network in the given city.*

BikeViz presents the user with a list of cities to choose from when entering the website,

with a view of Paris as default. The user can select five cities for display on the map: Paris, Copenhagen, Manchester, Munich, and Tokyo, as seen in figure 9. As the user clicks on a city, the map navigates to the given city and highlights the selected city button to indicate which city the user is currently viewing. When navigating to a new city, the bicycle network layers are reset to their default values, with the synthetic network and slider active and the slider stage set to 0 to encourage interaction.

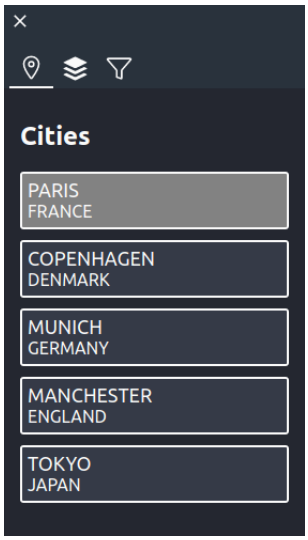


Figure 9: Sidebar: Cities Tab

Functional Requirement 3 *As a user, I want to be able to choose between the existing and the synthetic bicycle network because I want to compare the two.*

The user can choose different layers to display on the map in the layers tab, depicted by a layer icon in the sidebar. The default layer is the synthetic bicycle network. The layers tab include checkboxes used to toggle between the different layers, including the choice to display the existing bicycle network either on top of the synthetic network or by itself. Additionally, the slider has a checkbox to allow the user to display the synthetic network without the slider. When toggling off the synthetic network checkbox, the slider checkbox will be disabled to ensure that the slider is not active when the network is not displayed.

Clicking the 'reset layers' button resets all layers to their default position. The layers tab can be seen in figure 10.

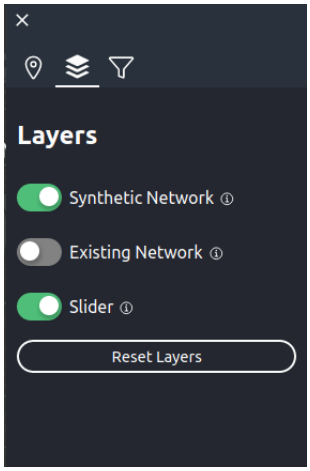


Figure 10: Sidebar: Layers Tab

Functional Requirement 4 *As a user, I want to be able to customize the visualization of the map design because I want to discover the bicycle networks in different layouts.*

The filter tab in the sidebar, represented with a funnel icon, provides access to customization options for the map. The funnel icon is associated with filtering and commonly used across the web for the same purpose. Toggle buttons represent the modification options of the layout, with associated labels describing the customization. The filters tab can be seen in figure 11.

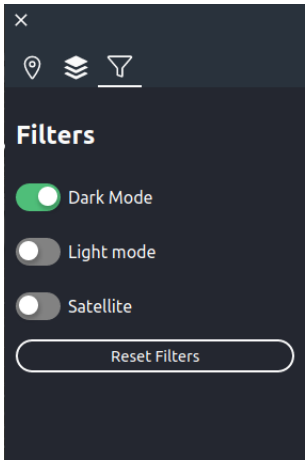


Figure 11: Sidebar: Filters Tab

The application initially had two toggle options, providing the user with three different map layouts: dark, light, and satellite. 'Light mode' would be applied if 'dark mode' was toggled off. However, to align the number of options to reflect the number of layouts, three toggle options are available: dark mode, light mode, and satellite. Toggling either 'Light Mode' or 'Dark Mode' automatically switches the opposite to demonstrate which mode is active and deactivated visually.

Dark mode displays a dark-colored map as known from the graphical interfaces of various global positioning systems (GPS) at night, it is the default toggled option for the application. Light mode displays a light-colored map as known from Google Maps and satellite mode displays a real-life depiction of the map as captured by satellites. The different map styles can be seen in figure 12.

Figure 12: Paris in Dark, Light, and Satellite Mode

Functional Requirement 5 *As a user, I want to see statistics regarding the altered data metrics of the synthetic bicycle network because I want to see the progression at each stage.*

The statistics window can be accessed in the top right corner of the screen by clicking the bar chart icon. The window's heading includes the selected city, the current stage of the algorithmically grown network, and the slider value. Furthermore, it displays relevant data metrics for the growing bicycle network that dynamically updates according to the stage of the network currently depicted on the map.

The metrics include the length of bicycle lanes in kilometers, a price estimate, coverage of the bicycle network in km^2, and the increase or decrease in percentages for the directness of the largest connected component (LCC), efficiency local, and efficiency global. The statistics

window can be closed by clicking the cross in the top left corner of the window or clicking another navigation icon. Furthermore, when turning off the synthetic network in the layer tab, the statistics window automatically closes. A depiction of the statistics window can be seen in figure 22 in section 6.4: The User Interface.

Functional Requirement 6 *As a user, I want the freedom to be able to interactively navigate between the different stages of the synthetic bicycle network to explore how it expands.*

The implementation of the slider is to enable interactive navigation between the different stages of the synthetic network. The final version of the slider can be seen in figure 13.

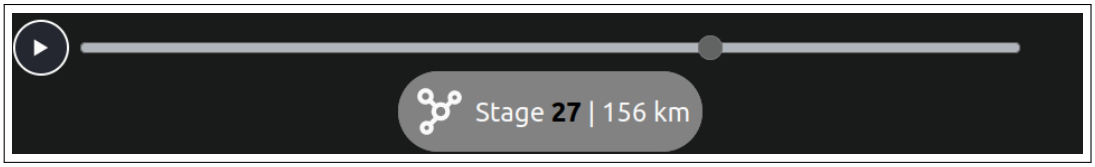


Figure 13: Final Slider

The slider has 40 option values to represent the 40 stages of each city's network. The slider's visual design is a long narrow bar, topped with a circle positioned at the slider's current value, ultimately reflecting the progression stage of the network. Below the slider is a display notifying the user of the current stage value. The slider can be dragged by pressing and holding the left mouse button while dragging it, which changes the stage, or by clicking on the bar and using the keyboard's arrows to increment or decrement the slider value.

Initially, the display consisted of a pin icon, and a paragraph with the current stage, as seen in figure 14.

Figure 14: Initial Display Stage

In order to more clearly convey what a stage reflects, the icon was changed to a graph depicting an expansion of graph nodes, hoping to achieve a more accurate depiction of a growing network. Furthermore, a length metric in kilometers was added to clarify that the network is expanding in size at each stage. As there are now two numbers depicted, it may confuse the user which of the two is more important. Therefore, the stage number is emphasized to reflect that stage has precedence over kilometers. The result is depicted in figure 15. The changes were made based on feedback from our user.

Figure 15: Display Stage and Length of Grown Network in Kilometers

Functional Requirement 7 *As a user, I want to be able to access documentation on how to navigate BikeViz, to gain in-depth knowledge on how to utilize it.*

Information regarding the purpose and usage of BikeViz and documentation on the functionality and how to extend it can be accessed on a separate page, namely the "Documentation" page. The page is accessible from the main page of the application via a document icon working as a link to the page, as seen in figure 16. The icon is always visible and located on the right side of the screen.

Figure 16: The Navigation Menu

The link opens up a new tab to avoid overriding the current map view and styling, or in other words, the progress that the user has made within the application.

The landing page of documentation is a "Table of Contents" from which the user can navigate to the documentation's relevant sections, as seen in figure 17. On each page, an introduction with an explanation of the usage is provided, followed by detailed documentation on extending the corresponding functionality, as outlined in Appendix F, increasing in difficulty as the user progresses through each page.

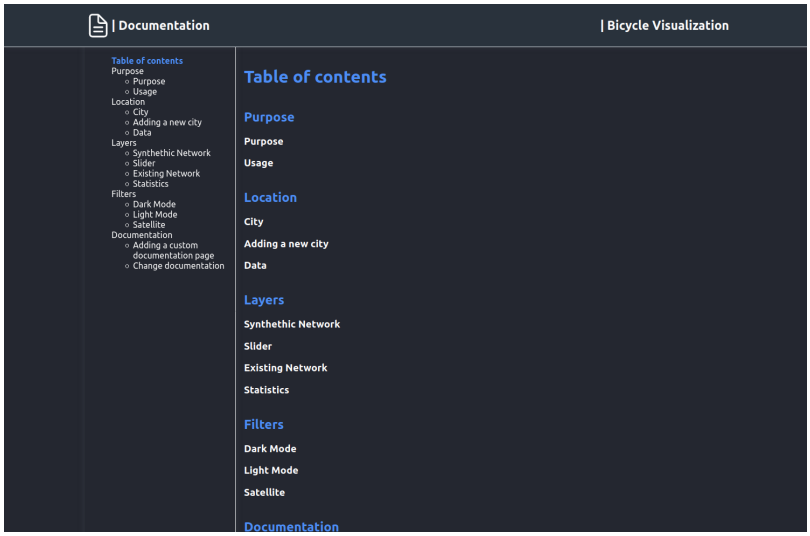


Figure 17: Documentation page - table of contents

The naming convention of the sections follows the functionality of BikeViz, so the user can easily recognize elements from the exploration of BikeViz's graphical user interface.

Functional Requirement 8 *As a user, I want to be able to see the core functionality of BikeViz when I enter the website, to engage with the application immediately.*

As the bicycle network data is loaded and the loading screen fades, BikeViz greets the user with a map of Paris. Instantaneously the slider starts auto-playing, incrementally revealing the algorithmically grown bicycle network of Paris. On the right side of the screen, an information window is displayed, informing the user about the core functionality of BikeViz, with text and icons. The left side of the screen holds the sidebar, informing the user of

the current city and country on display and providing the option to choose between other cities. A depiction of the core functionality which greets the user can be seen in figure 18.

Figure 18: The landing page of BikeViz

The autoplay function of the slider is initially depicted as a pause icon to indicate that something is playing. Hovering the button will color the background white, indicating an action when pressing the button. Clicking the button or manually dragging the slider will pause the autoplay function and change the button's appearance to a play icon. When the slider reaches its maximum value, the button will be replaced with a replay icon, clicking the icon results in the animation starting over from stage 0. The different buttons styles are pictured in figure 19.

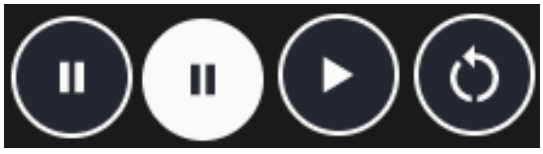


Figure 19: Pause/Play/Replay Buttons

As Usability Heuristics are concerned with the application's usability and typically used as evaluation metrics of the design, they are summarized in the following section. In contrast, the Gestalt Principles are concerned with the application's visual design and are evaluated and elaborated upon in section 6.4: The User Interface. Section 7: Reflections, Limitations, and Outlook elaborate on functional requirement 9, 10 and 11, as they have not been implemented in BikeViz as of yet, partly due to their prioritization as either should-have or could-have initiatives.

6.3 Implementing Usability Heuristics

This section expands upon the incorporation of Usability Heuristics. The Usability Heuristics designed by Jakob Nielsen are used as general guidelines to ensure that the application provides a user-friendly experience.

#1 Visibility of system status: The outline of borders around user-selected icons and the change of color applied to selected city buttons communicates the system's status to the user. Additionally, when the user clicks on a new city button, the inbuilt Mapbox function - `flyTo()` – redirects the user to a new city while keeping the user's attention. In connection to the slider, the display beneath the component dynamically updates as the user interacts with the slider, reflecting the current stage of the synthetic network on the map. Lastly, the grey and green coloring of the checkboxes in the layer and filter components indicates which layers or filters are on/off.

#2 Match between system and the real world: The choice of icons in BikeViz adheres to familiar map icons and recognizable icons such as the play/pause/replay button used to control the network animation. Furthermore, the slider's horizontal layout mimics a spectrum or timeline, as moving left is associated with a decrease in value and moving right simulates an increase.

#3 User control and freedom: For each component placed in the foreground of the map, the user can exit the visible component using a cross icon displayed in the top left corner, providing the user with a choice of what to display. In the layer and filter tabs, the reset buttons allow the user to reset all checkboxes back to the default position rather than doing

it manually. Moreover, the documentation page opens in a new tab to persist user progress on the map and incorporates buttons to help navigate the user to the previous or next pages. The filter tap provides the user with the choice of displaying the underlying map in either dark-, light - or satellite mode to cater to user preferences. Lastly, the user has the freedom to stop the synthetic network animation at any point in time, either by pressing the play/-pause icon or pressing anywhere on the slider.

#4 Consistency and standards: The underlying map in BikeViz adheres to standard map conventions, such as zooming in and out and dragging the map, assumed to be familiar to the user from other map applications and tools, such as Google Maps. Concerning consistency, in the documentation page, the naming of the pages reflects the same as the features in BikeViz.

#5 Error prevention: Whenever the synthetic network view is toggled off in the layers tab, the slider toggle bar is disabled to ensure that the user does not interact with the slider. The same applies to the filters tab, where the toggle buttons for dark- and light mode toggles opposite each other.

#6 Recognition rather than recall: The information window provides information on navigating the application using the same icons as the sidebar and navigation bar. Furthermore, each city button displays both a city's name and country to enable recognition rather than recall. Lastly, the toggle bars indicate what and if a network layer is displayed on the map or turned off, using green and grey coloring.

#7 Flexibility and efficiency of use: The slider is created as an input HTML tag, enabling the use of keyboard arrows to change the slider stage rather than dragging it.

#8 Aesthetic and minimalist design: The design of the overall user interface and each distinct interface element integrates the 8th principle to ensure a simple user interface that only depicts the information and functionality deemed relevant for the user.

#10 Help and documentation: Tooltips are displayed for each icon in BikeViz, to inform the user of the tab name before clicking. At the same time, the information window

contents intend to help users navigate the core functionality of BikeViz when entering the application. In addition, help and documentation can be found on the documentation page to assist the users further.

The 10 Usability Heuristics are described in Appendix D. The Usability Heuristics are regarded as best practice for usability and are applied to make BikeViz instinctive to navigate for the users[30]. The following section explores the incorporation of the Gestalt Principles in the user interface design as a whole.

6.4 The User Interface

The following section presents the user interface and its components to provide a thorough description of how users can navigate BikeViz and utilize the distinct features. The section assesses the implementation of an intuitive user interface, based on the Gestalt Principles, found in Appendix C, Figma prototype v2 in section 5.5: Figma Prototypes and feedback from our user. The section compromises an assessment of the user interface as a whole, followed by a focus on the individual components.

As portrayed in figure 20, the user interface consists of a sidebar on the left side of the page, a navigation bar at the top right, along with an information window, and a slider in the bottom center. The positioning of the interface components adheres to common map characteristics to relate to users' familiarity with other map applications. Initially, the landing page depicted the statistics window when entering BikeViz. However, to adhere to non-technical users, the final landing page depicts the information window to provide users with a brief introduction to BikeViz and its functionality.

The user interface implements the common region principle by grouping related elements in the same screen area, as in the sidebar component, which contains the functionality concerned with visualizing the network data and customizing the map. In contrast, the navigation bar contains the features related to the growth of the network over time, the insertion of new cities, and general information about BikeViz and its purpose. The slider component is in charge of depicting network growth and is positioned in the bottom center

[30] https://www.nngroup.com/articles/ten-usability-heuristics/

of the screen to encourage the user to interact with the data.

The figure-ground principle applies for all the abovementioned components by putting the sidebar, navigation bar, and slider component in the foreground to focus user attention on these specific components to encourage users to interact with the features.

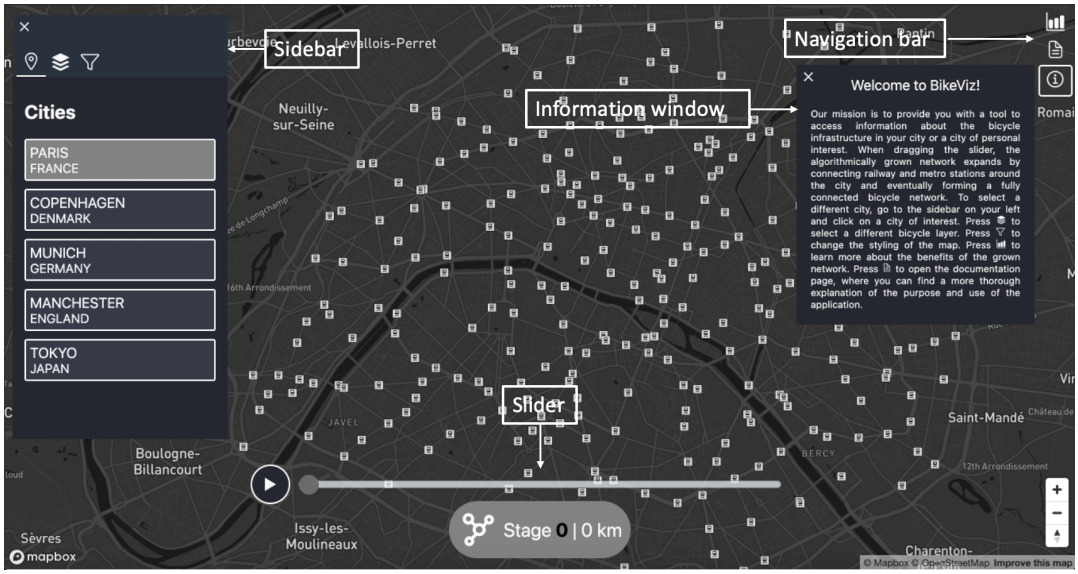


Figure 20: The User Interface

The Sidebar component consists of three tabs, the cities, layers and filters tabs that contain the logic for manipulating the map as presented in figure 21. The figure-ground principle applies by changing the background color on a city button when hovering on top of it and selecting it, indicating which city the user is currently viewing. The latter is further related to the focal point principle, as the selected city will stand out from the other cities. The similarity and proximity principles apply in all three tabs. However, they are most prominent in the cities tab, where city buttons are styled identically and placed next to each other, indicating shared functionality.

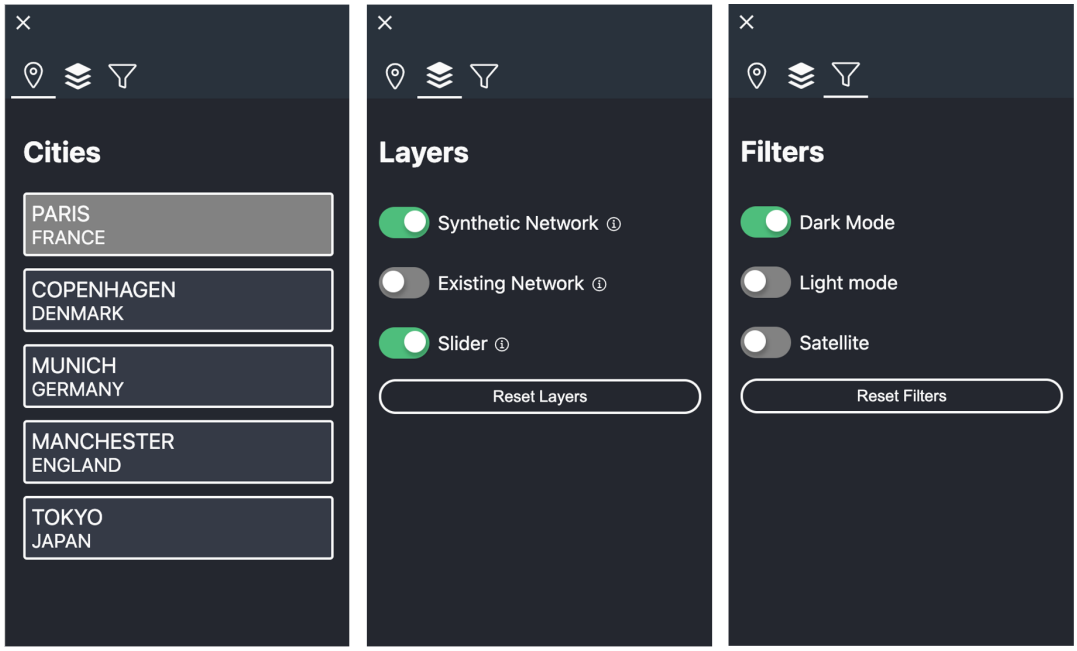


Figure 21: The Sidebar

The Navigation Bar holds the statistics, information, and documentation page components as pictured in figure 16 in section 6.2: Implementing Functional Requirements - Functional Requirement 7. The information window is displayed on the landing page as soon as a user enters BikeViz, providing the user with a quick explanation of the purpose and usage of BikeViz. The statistics window aims at the technical user by providing statistics about the growing bicycle network. In comparison, the documentation page accommodates a broader audience by providing information on usage for the everyday user and detailed information on extending functionality for the technical user. The figure-ground principle emerges when pressing on one of the icons by placing the selected component at the screen's forefront. The similarity principle is incorporated through the coloring of the icons to the opposite color of the map and by adding a border to the selected icon to indicate the user's selection. Proximity applies in the navigation bar by grouping together the elements concerned with the application's technical purposes.

The Information Window depicted in figure 8 in section 6.2: Implementing Functional Requirements - Functional Requirement 1, is displayed on the landing page of BikeViz to adhere to both set of users. The component provides a brief description of the purpose of BikeViz and how to navigate the user interface using text and icons similar to the ones in the sidebar and navigation bar. The component implements the figure-ground principle by placing the window in the screen's foreground, thereby focusing users attention on the window. Furthermore, it incorporates the focal point principle by using a larger font size for the heading *Welcome to BikeViz!* drawing the users attention to the window.

The Statistics Window is in charge of depicting relevant data metrics for the algorithmically grown bicycle network as it expands. The visual design of the component incorporates the principles of similarity and proximity, as the bar charts are positioned next to each other, with the same shape and size, indicating that they are percentages. Furthermore, placing the component on top of the map while dynamically updating values according to the stage of the network and slider further indicates an association with the grown network. A depiction of the statistics windows can be seen in figure 22.

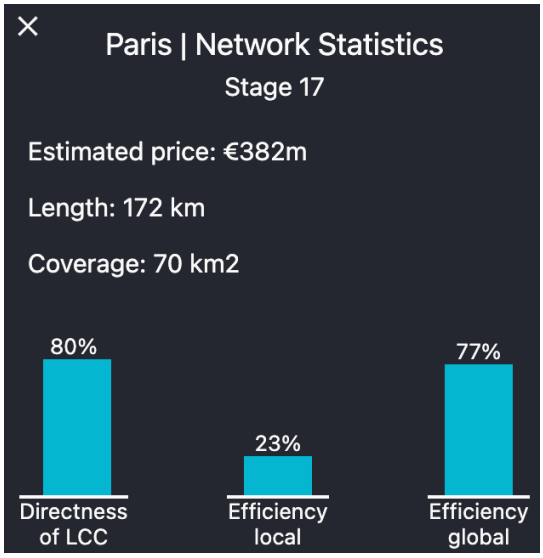


Figure 22: The Statistics Window

The Slider component is in charge of changing the algorithmically grown bicycle network stage when the user interacts with the slider. Moreover, it informs users of the current stage of the network and the number of kilometers added to the network at the current stage. The focal point principle applies by placing the slider in the center bottom of the screen to nudge the user to interact with the network. The principle of continuity emerges as the slider moves horizontally from left to right or vice versa while expanding or shrinking the network and changing the stage value accordingly. Additionally, the play/pause/replay button starts an animation of the network that automatically increments, taking advantage of the proximity principle, by placing the button to the slider's left, indicating that its functionality is related to the slider. The slider is pictured in figure 13 in section 6.2: Implementing Functional Requirements - Functional Requirement 6.

The Gestalt Principles adhere to principles of visual perception to ensure intuitive user interaction and familiar design in an application, presented in Appendix C. The Gestalt theory is based on the concept that the human brain attempts to simplify and structure composite images and design consisting of many elements by arranging the elements into an organized system that creates a whole[31]. Including the Gestalt Principles in the user interface can help improve the aesthetics of the design and ensure user-friendliness. The following sections elaborate on the limitations of BikeViz and the next steps to take in the expansion of the application.

7 Reflections, Limitations, and Outlook

This section includes the limitations found and reflections made through developing BikeViz and an outlook on the further development of the application.

7.1 The Limitations and Outlook of BikeViz

This section highlights the limitations of BikeViz found by evaluating the implementation of functional and non-functional requirements, along with limitations regarding the cost of building new bicycle lanes in different countries, responsive design, and the lack of usability testing. Additionally, the section puts forward the following steps to extend BikeViz and provide further usability and functionality.

[31] https://www.toptal.com/designers/ui/gestalt-principles-of-design

Functional Requirement 9: *As a user, I would like a step-by-step guide on how to navigate the application, in case I am uncertain of how to perform certain tasks.*
Initially, functional requirement 9 was planned as a tutorial on using and navigating BikeViz, as seen in both Figma prototypes. However, due to time constraints and prioritization of the must-have requirements, the requirement was not pursued. On the other hand, the design of BikeViz is based on standard design principles and usability heuristics to make it intuitive to users, with a documentation page and information window to guide the user further when using the application. With this functionality combined, going forward, it would be relevant to reevaluate the need for implementing a tutorial.

Functional Requirement 10: *As a user, I would like to compare two different cities simultaneously because I want to compare their existing or synthetic bicycle network.*
Functional requirement 10 is prioritized as a could-have initiative for the application. However, after consulting with our user, we found that the functionality for comparing two cities using dual screen is irrelevant as we assume that users are mainly concerned with the bicycle infrastructure and how to improve it in their city of residence.

Functional Requirement 11: *As a user, I want to be able to share the bicycle network view I am currently exploring with others because I want to convey the information I have obtained to people of interest.*
The requirement was prioritized as a could-have initiative, and therefore not implemented during the development of the Minimum Viable Product. However, the requirement is still relevant for the expansion of the application, to allow users to share the view of the bicycle network they are currently exploring with people of interest.

The documentation page: Going forward, the documentation page should include the functionality of a search engine for easier access to relevant documentation, adhering to usability heuristic #10. Furthermore, the documentation content should include help for users to recover from errors, adhering to usability heuristic #9.

Adding new bicycle networks: Besides the depiction of the algorithmically grown and the existing bicycle network, adding additional network layers to the map can be beneficial

to users as to compare different bicycle development strategies. This is especially relevant in cities with a limited number of metro and railway stations, where a grown bicycle network connected by grids might prove more valuable to users. Furthermore, other layers such as off-street bicycle lanes or shared bike and car mobility layers, can give more perspective on the overall bicycle infrastructure and its relation to other forms of transportation.

Fetching data from an external database: The application currently stores the bicycle network data for five different cities locally, occupying considerable space. When expanding BikeViz, it will be relevant to add more cities and additional bicycle network layers, which will result in a large amount of data to be stored. Rather than storing such large amounts of data locally, fetching the data from an external database may prove more efficient and make it easier to add additional data and organize it as the project expands.

Responsive design: BikeViz is only usable on laptops, desktops, and other large devices as the user interface do not incorporate a responsive design. Visualizing the synthetic or existing network poses problems on smaller devices as the networks cover entire cities, a view not feasible on smaller devices without zooming out, as only parts of the network stays visible as the network expands. However, zooming out on the map results in elements melting together, leaving paths indistinguishable from one another. Observing the city of Tokyo results in the same phenomenon, as seen in figure 23. The problem imposed requires an abstraction of elements depicted on the map going forward.

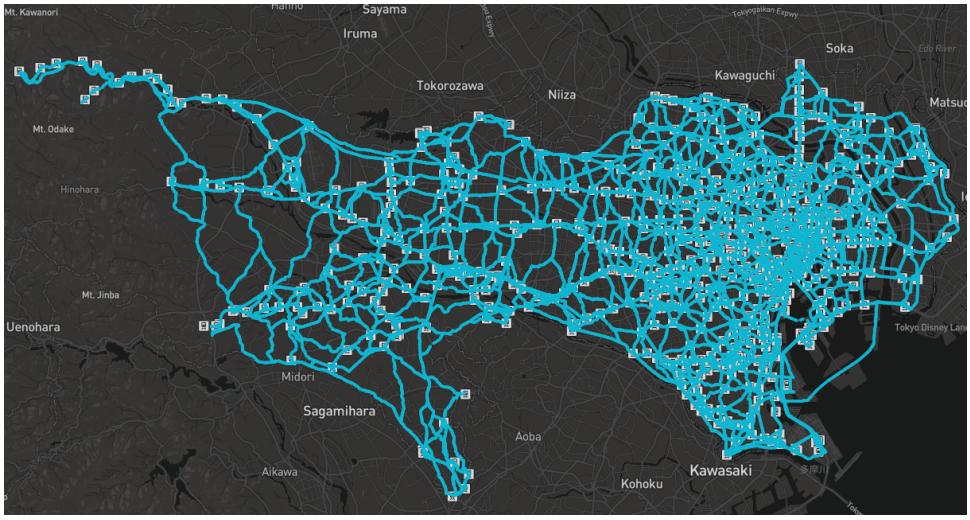


Figure 23: Tokyo with fully extended synthetic network

Costs of building segregated bicycle lanes: The cost of building *segregated* (see Definition of Concepts: Protected/Segregated bicycle lane) bicycle lanes varies between cities, countries and continents. Moreover, the costs associated with expanding the existing bicycle infrastructure are dependent on a variety of factors, such as the current state and width of the roads, other road users, city versus open land areas, and many more factors that differ from country to country. As reliable data for assessing the costs of building segregated bicycle lanes is not available for all of the implemented cities, an average estimated price per kilometer is calculated from available data in Copenhagen, Denmark[32]. In Denmark, the average cost per kilometer of building a one-way bicycle track on each side of the road, including drainage, costs between 5.5-16.5Mio/km DKK in large cities. Since taking the minimum and the maximum cost for one kilometer of bicycle track and calculating the average will be biased depending on the distribution, the maximum price of 16.5Mio/km DKK was chosen as the price estimate to account for all possible costs related to building new bicycle lanes. This price estimate is multiplied by the expansion in length (kilometers) at each stage of the grown network, displayed alongside the other statistics. For all countries, except Denmark, the price appears in euros. Going forward, it will prove beneficial to gather an estimate of the average cost per kilometer of protected bicycle track build in

[32] `https://idekatalogforcykeltrafik.dk/overslag-og-prisberegninger/`

the various cities or countries to provide users with a more representative estimated cost of expanding the bicycle network in their city.

Usability Testing: Usability testing is helpful to inform developers if certain functionality needs to be adjusted or proves irrelevant to the needs of the user[5] and can be carried out at several stages during the development of a product, such as the Figma prototypes, the deployed web application, and when adding new features. However, due to the time constraint of writing the thesis and the prioritization of developing a web application ready for deployment, usability testing has not been performed. Usability testing of the prototype should eventually be conducted on defined users to analyze the application for usability problems. This involves testing BikeViz on users to assess if they run into incomprehensible tasks while navigating the application and the user interface.

7.2 Reflection on the process in light of the Interaction Design Lifecycle Model

This section brings forward reflections acquired during the project using the model presented in section 5: Methodology: A simple Interaction Design Lifecycle Model. The model emphasizes a user-centered approach and the four basic activities of interaction design to include.

Point of departure: In the first sections of the thesis, we illuminate the advantages and disadvantages of different visualization frameworks. The majority of this work has been done prior to our master thesis in a research project[33] that established the foundation for our thesis. Early in the process of our thesis, we decided to continue building upon that foundation. Only a snippet of the prior research is presented in this thesis, and the selection of frameworks may come across as shallow. Summarizing a snippet of our prior research provides the reader with insights into the point of departure from which this project originated. If time were an irrelevant factor, it would be beneficial to apply the lifecycle model and reevaluate the selection of frameworks.

[33] https://www.overleaf.com/read/rvmbczypvvnk

The user-centered approach: The user-centered approach was carried out by creating two user personas based on limited empirical data gathered about the users. Alternatively, the design of BikeViz has mainly been tested on our supervisor and acquaintances.
Observing the current state of BikeViz as of now, it becomes clear that it is more characterized by fulfilling the use case scenarios of user persona 1 rather than user persona 2, in section 5.2.1: User Personas. We believe this is partly due to the technical nature of the synthetic network data and the understanding required to depict and gain information from the visualized data. The use case and gap that BikeViz attempts to fulfil concerning user persona 1, has been more exciting to accomplish. Whereas the simplicity of user persona 2's requirements and obtaining those requirements have been less attractive due to the existence of other tools already being able to meet such requirements. Future iteration cycles should accommodate the latter requirements to balance the target audience's weight, supported by the involvement of actual users to create an empirical foundation for the design.

The four basic activities: The first activity, concerned with identifying functional and non-functional requirements, was made with a point of departure in the user-personas, coupled with an ongoing correspondence with our supervisor, who served both as a user and stakeholder.

The implementation of functional and non-functional requirements make up BikeViz and is a continuous iterative process. The process involved redefining the requirements, adjusting them based on feedback, partial implementations in section 6: Developing the Minimum Viable Product, and the inevitable exclusion of some, as described in section 7.1: The Limitations and Outlook of BikeViz, before reiterating the entire process.

The Figma mock-ups, presented in section 5.5: Figma Prototypes, reflects the second activity in the model of designing alternatives and lays the foundation for the final prototype of BikeViz. When comparing them, it becomes apparent that the styling and design choices have changed quite a bit. The design of prototype v2 further builds on the Gestalt Principles in section 6.4: The User Interface, and Usability Heuristics presented in section 6.3 Implementing Usability Heuristics, with inspiration drawn from the literature on best map practices and similar map tools. Going forward, it would be beneficial to test the user experience of the user interface design by conducting empirical research on a larger audience and conducting usability testing.

The third activity of prototyping embodies the entire lifecycle model. The coding process began in the research project by establishing the core functionality of BikeViz, including projecting the map, displaying a synthetic network, and interactively engaging with the network. Functionality that adheres to a mixture of the functional and non-functional requirements enlisted in this report. Since then, the prioritization of features has followed the MoSCoW notation, which has functioned as a project management tool for deciding which feature to implement next. However, the prioritization does not take the non-functional requirements they embody into account and does not reflect the time spent implementing them, which has led to the postponing and exclusion of certain features.

The fourth activity, concerned with the evaluation of BikeViz, has been a continuous process involving the user-personas, supervisor reviews, and the estimated adherence to the Gestalt Principles and Usability Heuristics. Due to a lack of usability testing, user-personas were used to verify that the application fulfils the established requirements, leading to a reevaluation of the prototype. In the process, it has become apparent that objectivity cannot be achieved entirely through role-playing, which can be observed throughout the application in; the design, functionality and content, all characterized by our technical background and adhering to a more advanced user. The supervision reviews helped in prioritizing functionality and design alternatives. At the same time, the fundamentals of the Gestalt Principles and Usability Heuristics has functioned as the primary evaluation tool of the user interface.

Finally, as the four activities of the Simple Interaction Design Lifecycle model are all closely related, it becomes difficult to describe the reflections and categorize them as individual occurrences of one activity. Developing a product like an application may never result in a final product but merely an instantaneous snapshot of the current product. An application can continually be improved and optimized through testing and maintenance, thus moving forward, the reflections shall support us in this process.

The reflections made about the shortcomings of BikeViz enable us to create a plan for estimating the most crucial steps to take next. Furthermore, it helps us understand the limitations to the application and the process and journey that made us arrive at the current stage of BikeViz.

8 Conclusion

This thesis described the implementation and deployment of a web application, BikeViz, designed to visualize an algorithmically grown bicycle network connected by railway and metro stations in five cities. The thesis delineated the creation of BikeViz, starting from the motivation for developing the application, continuing with an introduction to previously conducted research, providing the basis for developing the application. The process of adhering to the Simple Interaction Lifecycle Model followed, by defining users and user scenarios, identifying functional and non-functional requirements and the actual implementation of the user interface and related functionality. Lastly, the thesis outlined the evaluation process of BikeViz, through the limitations of the application, its future expansion and the reflections made through applying the Simple Interaction Lifecycle Model.

BikeViz successfully implements all functional requirements listed as must-have initiatives according to the MoSCoW notation and one should-have initiative. The remaining requirements are either partially incorporated into the must-have requirements or regarded as future functionality. The thesis reflects and proposes implementation guidelines for the remaining requirements, including one should-have and two could-have initiatives. The visual design of BikeViz incorporates the Gestalt Principles to capture the user's attention and ensure an intuitive design. At the same time, Jakob Nielsen's Usability Heuristics guided the functionality of the user interface components of BikeViz, to enhance and evaluate the usability of the user interface. Both principles provided valuable guidance in the design of BikeViz.

At the beginning of the thesis, the bicycle data for the algorithmically grown network consisted of 40 separate files. In an attempt to format the data and reduce the total size, the data wrangling procedure lead to a reduction in file size from 14.5 Mbit to 8.5 Mbit for Paris and the elimination of duplicate LineString coordinates in the network data. As a result, the fetching of data improved with a speed of *21666%* and thereby and average decrease of *1607.65 ms*, which also includes the data for the existing bicycle network, points of interest and statistics. The remaining cities apply the same data-wrangling method.

As a result, BikeViz successfully fulfils the intended impact of providing users with the ability to visually compare the algorithmically grown bicycle network with the existing network. Moreover, comparing the two networks can assist users in identifying where

to place concrete bicycle lanes in a city to efficiently expand the existing network. Furthermore, studying the existing network can assist users to discover disconnected bicycle network components in a city and dissimilarities in the bicycle infrastructure between city areas. Expanding BikeViz and adding other algorithmically grown networks based on different growth strategies will allow users to compare bicycle development strategies to find the most suitable for their city.

One major limitation of BikeViz is the limited involvement of the actual users the application targets. A user-centered approach requires users' continuous involvement to guide the design and ensure that the application meets their needs. In the future, continuous usability testing should be conducted on the targeted users to ensure the utilization of BikeViz.

References

[1] R. Kearns, D. Rees, A. Macmillan, J. Connor,, K. Witten, and A. Woodward. The societal costs and benefits of commuter bicycling: Simulating the effects of specific policies using system dynamics modeling. *Environmental Health Perspectives*, 122(4), 2014.

[2] CROW. *Design manual for bicycle traffic.* 2016.

[3] D. Zuev K. Psarikidou and C. Popan. Sustainable cycling futures: can cycling be the future? *Applied Mobilities*, 5(3), 2020.

[4] G. Iñiguez L. G. Natera Orozco, F. Battistion and M. Szell. Data-driven strategies for optimal bicycle network growth. *Royal Society Open Science*, 7(12), 2020.

[5] Soren Lauesen. User interface design: A software engineering perspective. pages 413–441. Addison-Wesley, 2004.

[6] Munk-Madsen A. Nielsen P. Mathiassen, L. and J. Stage. *Object-oriented Analysis and Design*, page 119. Marko, 2000.

[7] T. Nagel. Visually analysing urban mobility: Results and insights from three student research projects. *KN - Journal of Cartography and Geographic Information*, pages 11–18, 2020.

[8] Till Nagel, Joris Klerkx, Andrew Vande Moere, and Erik Duval. Unfolding – a library for interactive maps. In Andreas Holzinger, Martina Ziefle, Martin Hitz, and Matjaž Debevc, editors, *Human Factors in Computing and Informatics*, pages 497–513, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[9] Sharp-H. Preece, J. and Y. Rogers. *Interaction Design: Beyond Human-Computer Interaction 4th edition.* International Series of Monographs on Physics. John Wiley Sons, 2015.

[10] R. Lovelace R. Aldred, T. Watson and J. Woodcock. Barriers to investing in cycling: Stakeholder views from england. *Transportation Research Part A: Policy and Practice*, 128, 2019.

Appendix A: datawrangler.py

```
import os
import json
from geojsonConverter import write_to_geojson

# Should use dataForWrangling dir instead.
dirPath = "../dataForWrangling/ParisRailWay"
os.chdir(dirPath)

def read_json_file(file_path):
    with open(file_path) as json_file:
        data = json.load(json_file)
        return data

files = os.listdir()
files.sort(key=lambda f: os.stat(f).st_size, reverse=False)


# Dict to store all ID's
allDict = {}
counterElse = 0
counterIf = 0
arrayOfDictStages = []

for file in files:
    if file.endswith(".json"):
        file_path = f"{file}"

        stageDict = {}

        data = read_json_file(file_path)
        for obj in data["geometries"]: # Check for objects in the column "geometries"
            if obj["type"] == "LineString": # Only add if type "LineString"
                minLat = min(obj["coordinates"])
                maxLat = max(obj["coordinates"])

                id = f'{minLat}{maxLat}'

                if id not in allDict:
                    allDict[id] = obj["coordinates"]
```

```
                    stageDict[id] = obj["coordinates"]

        arrayOfDictStages.append(stageDict)


# Convert each element in arrayOfDictStages to one GeoJSON file, with new property to
    differ between stages

write_to_geojson(arrayOfDictStages)
```

Appendix B: geojsonConverter.py

```
import os
import json
import typing

def read_json_file(file_path):
    with open(file_path) as json_file:
        data = json.load(json_file)
        return data

def create_geojson_dict(arr: list) -> dict:
    gjsonDict = {}
    gjsonDict["type"] = "FeatureCollection"
    gjsonDict["features"] = []

    for count, array_ele in enumerate(arr):
        stage_arr = []
        for key in array_ele:
            value = array_ele.get(key)
            stage_arr.append({"type" : "LineString", "coordinates" : value})

        gjsonDict["features"].append([{"type" : "Feature", "properties" : {"stage" :
             count}, "geometry" : {"type": "GeometryCollection", "geometries" : stage_arr
             }}] )
    return gjsonDict

def write_to_geojson(arr: list):
    with open('PARIS.json', 'w') as outfile:
        json.dump(create_geojson_dict(arr), outfile, indent=4)
```

Appendix C: Gestalt Principles of Design

The seven Gestalt Principles[34]:

1. **Figure-ground** relates to the way the human brain distinguishes objects it considers to be in the foreground of an image and the background. The figure-ground principle is useful when designers want to focus users attention towards a specific feature or call to action.
2. **Similarity** revolves around tying elements together that are positioned apart from each other in a design, by grouping elements by color, shape, or size.
3. **Proximity** relates to the grouping of elements, as elements positioned beside each other appear to be more related than elements spaced apart.
4. **Common region** is closely related to the proximity principle. Human's perceive objects positioned within the same close area as being grouped, regardless of their shape, color, proximity, etc.
5. **Continuity** states that elements arranged on a line or curve are perceived to be more related than elements not on the line or curve, as the human perception will follow the simplest path.
6. **Closure** is concerned with the idea that the human brain will fill in the blanks of a design or image to create a whole picture.
7. **Focal point** states that elements that stand out visually will capture and hold the user's attention first.

[34] https://www.toptal.com/designers/ui/gestalt-principles-of-design

Appendix D: Usability Heuristics by Jakob Nielsen

The 10 Usability Heuristics by Jakob Nielsen[35]:

1. **Visibility of system status**: The design should always keep users informed about what is happening through appropriate feedback within a reasonable amount of time.
2. **Match between system and the real world**: The design should speak the users' language. Use words, phrases, and concepts familiar to the user, rather than internal jargon. Follow real-world conventions, making information appear in a natural and logical order.
3. **User control and freedom**: Users often perform actions by mistake. They need a clearly marked "emergency exit" to leave the unwanted action without going through an extended process.
4. **Consistency and standards**: Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform[36] and industry conventions.
5. **Error prevention**: Good error messages are essential, but the best designs carefully prevent problems from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before committing to the action.
6. **Recognition rather than recall**: Minimize the user's memory load by making elements, actions, and options visible. The user should not have to remember information from one part of the interface to another. Information required to use the design (e.g. field labels or menu items) should be visible or easily retrievable when needed.
7. **Flexibility and efficiency of use**: Shortcuts — hidden from novice users — may speed up the interaction for the expert user such that the design can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

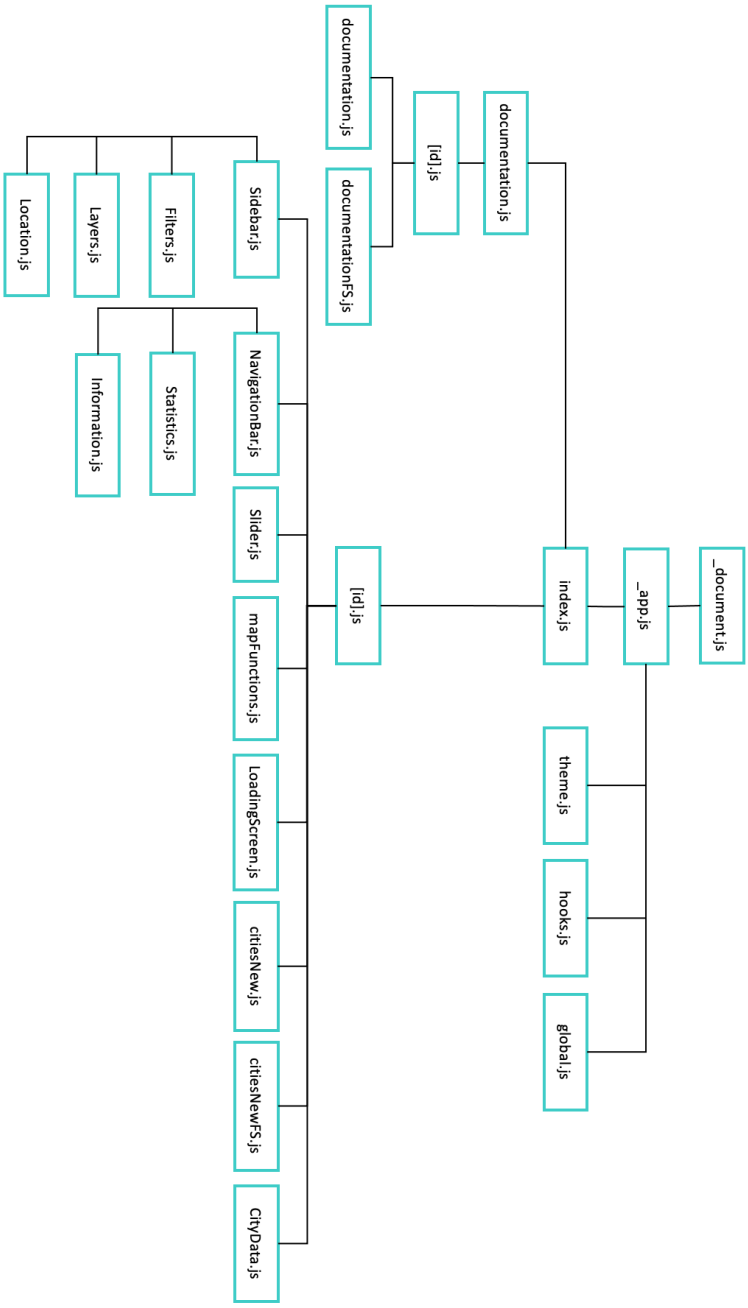
[35] https://www.nngroup.com/articles/ten-usability-heuristics/
[36] https://www.nngroup.com/articles/do-interface-standards-stifle-design-creativity/

8. **Aesthetic and minimalist design**: Interfaces should not contain irrelevant or rarely needed information. Every extra unit of information in an interface competes with the relevant information units and diminishes their relative visibility.

9. **Help users recognize, diagnose, and recover from errors**: Error messages should be expressed in plain language (no error codes), precisely indicate the problem, and constructively suggest a solution.

10. **Help and documentation**: It is best if the system does not need any additional explanation. However, it may be necessary to provide documentation to help users understand how to complete their tasks.

Appendix E: Class Diagram



Appendix F: Pipeline Instructions for Adding Additional Data

Pipeline for adding a new city

Appendix F thoroughly explains how to extend the currently implemented functionality of the application, including cities, layers, and documentation. The purpose of the appendix is to guide developers in the extension of the functionality of BikeViz, in an attempt to inspire people to expand the project and enrich it with their knowledge.

Adding a new city to the map requires a few prerequisites:

1. The 39 JSON files that make up the algorithmically grown bicycle network for the given city and an additional JSON file that makes up stage 0, containing an empty GeometryCollection. The file should only contain the text displayed below.

```
{"type": "GeometryCollection", "geometries": []}
```

   The algorithmically grown bicycle network now consists of 40 files in total.

2. A JSON file containing the coordinates for the existing bicycle network.

3. A JSON file containing the data metrics for the algorithmically grown bicycle network. It is important to add an extra array entry in the JSON file to represent stage 0. The entry should contain the same data metrics as the other array entries. However, all values are set to 0, as nothing happens in stage 0. The array entry should look as follows:

```
{
"length": 0,
"length_lcc": 0,
"coverage": 0,
"directness": 0,
"directness_lcc": 0,
"poi_coverage": 0,
"components": 0,
"overlap_biketrack": 0,
"overlap_bikeable": 0,
"efficiency_global": 0,
```

```
12      "efficiency_local": 0,
13      "efficiency_global_routed": 0,
14      "efficiency_local_routed": 0
15      }
```

When the above mentioned steps has been completed, it is time to insert the data into the application.

1. Navigate to the Cities folder inside nextjs/public/cities and create a new city directory, call the directory the name of the city in lower case.
2. Navigate into the newly created directory and add 4 additional directories called:
   (a) *city*
   (b) existing
   (c) statistics
   (d) *city*Markers

   Instead of *city* insert the city you are adding to the map.
3. Next, navigate to the dataForWrangling directory and create a new folder called *City*RailWay, with the name of the city to insert and the first letter in uppercase
4. Now it is time to insert the data into the correct directories as follows:
   (a) In the existing directory, insert the JSON file for the existing bicycle network in the city, rename the file *city*biketrack.json.
   (b) In the *city*Markers directory, insert the JSON files for the points of interest, made up of railway and metro stations. Rename the file *city*_poi_railwaystation.json.
   (c) In the statistics directory, insert the JSON file containing the data metrics for the different stages, rename the file to *city*.json.
   (d) In the *city*RailWay directory inside the dataForWrangling folder, insert the 40 JSON files that makes up the algorithmically grown bicycle network.

5. The data for the algorithmically grown bicycle network, located in the *city*RailWay directory has to be converted into a FeatureCollection with 40 different stages, to be displayed on the map. In order to so, take the following steps:

   (a) Navigate to the datawrangler.py class inside nextjs/utilities directory, and insert the path to the *city*RailWay folder on line 6, as such:

   ```
   1        dirPath = "../dataForWrangling/cityRailWay"
   ```

   Replace *city* with the name of the city to be added.

   (b) Next, navigate to the geojsonConverter.py class inside the nextjs/utilities directory. Go to line 50 and change it to the following:

   ```
   1        with open('CITY.json', 'w') as outfile:
   ```

   Replace *CITY* with the name of the city to be added in uppercase letters.

   (c) Open the terminal and navigate inside the utilities folder and run the following command: PYTHON3 DATAWRANGLER.PY. This will create a new file called *CITY*.json, inside the *city*RailWay directory in the dataForWrangling directory, drag the *CITY*.json file inside the newly created *city* directory, from step 2a.

   (d) Lastly, navigate to the CityData.js file inside the nextjs/public directory and add an array entry for the given city, as shown below:

   ```
   1        city: {
   2        city: "City",
   3        country: "Country",
   4        coordinates: [coordinate_1, coordinate_2],
   5        zoom: 11,
   6        }
   ```

   Replace the *city* with the name of the city and remember to put the first letter in uppercase on line 2 inside the quotation marks. Add the country's name on line 3 inside the quotation marks and put the first letter in uppercase. On line 4, put the coordinates for the center of the city to focus the map. On line 5, insert the zoom level for the given city.

The new city is now added to BikeViz. If it is not shown on the map and sidebar, close the application and rebuild it inside the terminal by running the command: YARN DEV.

Pipeline instructions for adding a new layer

The motivation for adding additional layers to the map is endless and only limited by one's imagination. The motivation to add more layers could be to expand upon the application's functionality to display other networks on the map, such as algorithmically grown networks between points of interest other than railway and metro stations.

To add a new layer, navigate inside the directory called `map` and the file `mapFunctions`, which contains the existing functions used for creating and applying layers to the map. The layers can be modified and styled in a plethora of ways. To study these, refer to Mapbox GL's API[37].

The first thing to do is create a new function in the file `mapFunctions`. The function should take a *map* as the only parameter. To ensure that the function can be reused by different maps if desired.

Inside the newly created function, add a source to the referenced map given in the parameter. The `addSource` function is an inbuilt Mapbox GL function, which takes an arbitrary name as the first parameter. The second parameter receives an object with the type of data, e.g. GeoJSON and a reference to the data. Set the data to null, and delegate the job of setting the data reference to another function. The reasoning behind this approach is for the data to be dynamic rather than static to avoid creating individual layers for data points.

Next, add the actual layer by referencing the map and calling the Mapbox function `addLayer`. The `addLayer` function takes a layer object. First, give it a unique *id*, then reference the source, which should be corresponding to the one created before with the `addSource` function. The *type* property specifies the type of layer and must be either background, fill, line, symbol, raster, circle, fill-extrusion, heatmap, hillshade or sky. Depending on the type of layer, specify either or both the *layout* and *paint* property. Below is an example of the code used to display the points of interest layer in BikeViz.

```
1 export function addSourcesAndLayers(map) {
2   map.addSource("poi", {
3     type: "geojson",
4     data: null,
5   });
```

[37] https://docs.mapbox.com/mapbox-gl-js/style-spec/layers/

```
  map.addLayer({
    id: "pointOfInterest",
    source: "poi",
    type: "symbol",
    layout: {
      "icon-image": "rail",
      "icon-padding": 0.0,
      "icon-allow-overlap": true,
      "icon-size": ["interpolate", ["linear"], ["zoom"], 7, 1, 15, 0.5],
    },
  });
}
```

As the function for adding the layer is in place, import it, add the layer to the map and set the layer's data.

The layer function is imported into the `city/[id].js` file. Inside the functional City component, create a `useEffect` for initializing the map layer. Navigate to the map `on` function, insert the newly created function and give it the initialized map as a parameter. The layer is now added to the map whenever it is loaded. However, in order for it to display the layer, it is necessary to set the data of the layer's source.

Still inside the functional City component, create a new `useEffect` to deal with the job of setting the data. Create a conditional statement inside the useEffect to check if the map is initialized, else return nothing. Then, create a variable to hold a reference to the map's layer source and with a conditional statement, check if the layer has been added to the map, else return nothing. This is done by calling the `getSource` function on the map reference, with the parameter as the name of the source. The data is then set by calling `setData` on the variable, with the data object as the parameter. Finally, give the `useEffect` a second argument, consisting of an array with the map and data to be set, this optimizes the performance, by only applying the `useEffect` when the map is loaded or when the data changes. The layer is now visible on the map. The code example presented below is in charge of displaying the icons for railway and metro stations on the map.

```
// Setting the "points of interest" data
 useEffect(() => {
   if (!isMapboxLoaded) return;
   const source = mapInstanceRef.current.getSource("poi");
```

```
    if (!source) return;
    source.setData(newData.pointer);
  }, [isMapboxLoaded, newData]);
```

Pipeline instructions for adding or changing documentation pages

Adding a custom documentation page

In order to add a custom dynamic route to the documentation pages, so the slug becomes:

bicyclevisualization.com/documentation/[id]

First, create a new markdown file with the information desired for display (see `https://www.markdownguide.org/basic-syntax/` for help with the markdown syntax). Then, change directory to `nextjs/documentation`, which contains the other markdown files and checkout their syntax for inspiration. Once inside the documentation directory, create a new markdown file and name it to reflect the file's contents. Open the newly created file and create a title, following this convention:

```
---
title: "{Same as the file name}"
tag1: "Header1"
tag2: "Header2"
tag3: "Header3"
---
{Your text ... }
```

This will display the title in the table of contents and the navigation menu. Afterwards, fill the file with the text for display.

The tags presented above are optional, as they function as headers. However, when adding content to the markdown file, it is advised that the tags reflect the text headers, e.g. if adding a paragraph with the following header:

Dark mode

It is best to add a tag following this convention:
`tag1: "Dark mode"`

Change documentation

Changing the current documentation is similar to the convention of *adding a custom documentation page*. Instead of creating a new markdown file, change the contents of one of the current files.