BERGISCHE
UNIVERSITÄT
WUPPERTAL

Web-based Exploratory Visualisation and Analysis of Pedestrian Simulations in Buildings

---

Master Thesis

LuFG Computersimulation für Brandschutz und Fußgängerverkehr
Fakultät 5 – Abteilung Bauingenieurwesen
Bergische Universität Wuppertal

Supervisors:
Prof. Dr. Armin Seyfried
Dr. Mohcine Chraibi

submitted by:
Tao Zhong 1327346

Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Wuppertal, November 3, 2020 *Tao Zhong*

Abstract

With the development of software engineering, several tools have been developed and improved for the visualization and analysis of evacuation simulations in buildings. However, the existing software for pedestrian traffic researches is usually dependent on a specific operating system platform or development kit. While it enables the developer to use the tool easily, it also limits the scenarios in which the tool can be applied. There are several technologies on the market that are designed to provide a cross-platform solution and a reliable architecture for software, making it easier to enhance the software itself.

With the goal of making cross-platform software for the visualization and analysis of pedestrian traffic more flexible, a new software is presented in this thesis. The new software is based on the development of modern technologies and can be run on different operating systems and devices. It also allows researchers to easily access the software without being restricted by devices or operating systems.

Functionally, the new software is able to visualize the geometry of buildings and the trajectories of pedestrians. For visualization, both 2D and 3D are implemented in the new software, so that researchers can switch between them as needed. In addition to the graphical layer, the software also allows the user to interact with the scene to better observe the geometry and pedestrians.

Finally, the analysis module is merged with the software to analyze explorative data from simulations and to generate fundamental diagrams. The new software allows the fundamental diagrams to be displayed directly, which helps researchers to better understand the process of simulation.

Kurzfassung

Mit der Entwicklung von Software-Engineering wurden einige Tools für die Visualisierung und Analyse von Evakuierungssimulationen in Gebäuden entwickelt und verbessert. Die bestehende Software für Fußgängerverkehrsforschung ist jedoch in der Regel von einer bestimmten Betriebssystemplattform oder einem Entwicklungskit abhängig. Es ermöglicht dem Entwickler zwar eine einfache Nutzung des Tools, schränkt aber auch die Szenarien ein, in denen das Tool eingesetzt werden kann. Es gibt verschiedene Technologien auf dem Markt, die darauf ausgerichtet sind, eine plattformübergreifende Lösung und eine zuverlässige Architektur für Software zu bieten, wodurch es einfacher wird, die Software selbst zu verbessern.

Mit dem Ziel, plattformübergreifende Software für die Visualisierung und Analyse des Fußgängerverkehrs flexibler zu gestalten, wird in dieser Abschlussarbeit eine neue Software vorgestellt. Die neue Software basiert auf der Entwicklung moderner Technologien und ist auf verschiedenen Betriebssystemen und Geräten lauffähig. Darüber hinaus ermöglicht sie Forschern einen einfachen Zugriff auf die Software, ohne durch Geräte oder Betriebssysteme eingeschränkt zu sein.

Funktionell ist die neue Software in der Lage, die Geometrie von Gebäuden und die Flugbahnen von Fussgängern zu visualisieren. Für die Visualisierung sind sowohl 2D als auch 3D in der neuen Software implementiert, so dass die Forscher je nach Bedarf zwischen beiden wechseln können. Zusätzlich zur grafischen Ebene ermöglicht die Software dem Benutzer auch die Interaktion mit der Szene, um die Geometrie und die Fußgänger besser beobachten zu können.

Schließlich wird das Analysemodul mit der Software zusammengeführt, um explorative Daten aus Simulationen zu analysieren und grundlegende Diagramme zu erstellen. Mit der neuen Software können die Fundamentaldiagramme direkt angezeigt werden, was den Forschern hilft, den Prozess der Simulation besser zu verstehen.

Contents

Declaration **iii**

Abstract **v**

Kurzfassung **vii**

Contents **ix**

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**

1.1 Motivation and objective 1

1.2 Outline 3

2 Scientific and Engineering Background **5**

2.1 Analysis tools for pedestrian dynamics 5

2.1.1 Observables 5

2.1.2 Fundamental diagram 7

2.1.3 Spatiotemporal profile 8

2.2 Visualization tools for pedestrian dynamics 9

2.3 Modern web technologies 11

3 Requirement and Solution Analysis **15**

3.1 Overview 15

3.2 Software engineering 15

3.2.1 User Interface 16

3.2.2 Server 18

3.3 Scientific research 19

3.3.1 Analysis 19
3.3.2 Visualization 23

4 JPSvis Online **25**
4.1 Architecture 25
4.2 Implementation 28
4.2.1 HTTP server 28
4.2.2 User Interface 32
4.2.3 3D visualization 36
4.2.4 2D visualization 48
4.2.5 Analysis module 52

5 Tests **55**
5.1 Test cases 55
5.2 Server hosting 56
5.3 Uploading files 57
5.4 Starting 3D view 59
5.5 Interacting with mouse in the 3D view 60
5.6 Interacting with menu in the 3D view 61
5.7 Starting 2D view 62
5.8 Interacting with mouse in the 2D view 63
5.9 Interacting with menu in the 2D view 64
5.10 Plotting diagram 65

6 Summary and Outlook **67**
6.1 Summary of JPSvis Online 67
6.2 Outlook 69

Bibliography **71**

List of Figures

2.1 Fundamental diagrams of unidirectional pedestrian flow 8
2.2 Spatiotemporal profiles of unidirectional flow 9
2.3 The User Interface of JPSvis 10
2.4 The User Interface of SumoViz 10
2.5 The User Interface of SumoViz3D 11

3.1 The User Interface draft - upload page 16
3.2 The User Interface draft - view page 17
3.3 A basic web server 18
3.4 Methods for calculating density and velocity 20

4.1 The Architecture of JPSvis Online 26
4.2 The tech stack of JPSvis Online 27
4.3 The sequence diagram of JPSvis Online 28
4.4 The activity diagram of $post\_file$ 32
4.5 The tag tree of the uploading page 33
4.6 The tag tree of the viewing page 35
4.7 Screenshots of the uploading page 37
4.8 The screenshot of the viewing page 38
4.9 Screenshots of the diagram window 38
4.10 Class diagram for 3D view 40
4.11 A geometry in 3D view 42
4.12 The pedestrian 3D model 43
4.13 The trajectory of pedestrians in 3D view 46
4.14 The menu for 3D view 46
4.15 The skeleton view of pedestrian model 47
4.16 Class diagram for 2D view 49
4.17 The 2D view 51

4.18 The density profile 54

5.1 The message in the terminal 56
5.2 Successful website opening 56
5.3 The message after successful uploading 58
5.4 The message after failed uploading 58
5.5 The 3D View 59
5.6 The 2D View 62
5.7 Plotting diagram 65

List of Tables

3.1 Output files with Method A 20
3.2 Output files with Method B 21
3.3 Output files with Method C 22

4.1 Affordable APIs 52
4.2 Outputs file for generating diagram 53

Chapter 1

Introduction

1.1 Motivation and objective

Since the population in urban areas is continuously growing[1], more and more high-rise and public buildings such as train stations, stadiums, airports, etc. have been constructed in recent decades. As a result, the potential safety risks have increased and will continue to increase as it is more difficult to evacuate the building with higher pedestrian density in a short time.

In response to such a challenge, researchers have developed a range of simulation tools to help urban planners, architects and organizers. Simulation software is often used within civil engineering projects by constructing virtual buildings and setting up various evacuation scenarios. In this way, the potential hazards in different scenarios can be effectively identified.

In most simulation programs the simulation results are usually stored in text format, since the simulation process consists in fact in calculating the values of the different characteristics of pedestrian behavior and generating the corresponding results. Obviously, solely text and number do not visually reflect the evacuation process. A visual representation of the simulation results is of great importance in such simulation software so that the safety designer can visualize the data, recognize patterns, identify anomalies and test and re-validate alternative designs.

JuPedSim is a toolkit developed by Forschungszentrum Jülich that provides a wide range of toolkits to fulfill the different requirements in evacuation simulation. As a module of JuPedSim, JPSvis is used to visualize the text data, transform it into

a 2D or 3D view and perform operations on the trajectory files to create various diagrams that help researchers in quantitative analysis. However, the current JPSvis is limited to desktop computers and cannot be used in mobile devices, which means that its application scenarios are limited by the performance and size of the computer hardware.

Given the limitations above, this thesis focuses on the development of a web-based software to replace the current visualization and analysis tool JPSvis in JuPedSim. As it runs in the browser, the new JPSvis is called "JPSvis Online" in this thesis. Not all features of JPSvis are implemented in JPSvis Online due to the limited scope of this thesis. Therefore, JPSvis Online is developed with the following objectives in mind.

The first objective is that JPSvis Online should be able to import geometry files created by the JPSeditor. Since the files contain information about buildings in a standardized format, namely rooms, stairs and other elements defined by x and y coordinates, JPSvis Online should be able to parse the geometry file and visualize it in a two-dimensional (2D) and three-dimensional (3D) view.

The second objective is that JPSvis Online should also be able to effectively parse the trajectory file created by JPScore that describes the movement of pedestrians in each frame during the simulation. Only with the geometry and trajectory files simultaneously can the evacuation simulation in buildings be rendered in JPSvis Online.

To better assist researchers, city planners and architects in reviewing and analyzing the evacuation process in the building, some additional information is needed in JPSvis Online in 2D and 3D views to reveal the additional properties such as pedestrian direction and speed.

As a web-based software, the way how users interact with the software must be considered comprehensively in JPSvis. No matter what the application scenario, be it the use of a desktop computer with mouse and keyboard, a laptop with trackpad or mobile devices with touchscreen, users should always be able to easily control and adjust the content in JPSvis Online.

The fifth goal is that JPSvis Online should be able to run a series of analysis scripts on the output file created by JPSreport to generate fundamental diagrams. At the

same time some useful information should be valid in the visualization area, e.g. the number of evacuated pedestrians.

The last objective focuses on the possible upgrades in the future. JPSvis Online should be designed with an extensible architecture in which the new components and features can be easily added. When JPSvis Online is developed as a web application, it should be deployed simply on the server so that users can visit it in real time via a browser.

1.2 Outline

This thesis mainly describes the design and implementation of JPSvis Online. In order to give a clearer overview of the content of this thesis, the outline follows here.

The second chapter reviews the background of pedestrian traffic, including the observations and diagrams frequently used in the simulation for analysis. Then the existing visualization tools are presented and their advantages and disadvantages are discussed in detail to clarify the requirements for JPSvis Online. Finally, the modern web technologies used in the development of JPSvis Online are also listed.

In the third chapter, the requirements for the JPSvis Online are further discussed in order to refine the objectives in the section 1.1. They consist of two parts: software engineering and scientific research. With focus on the functional components of JPSvis Online, the first part defines the functionalities of all components at the software engineering layer to ensure that JPSvis Online can perform properly. The second part clarifies the functions needed for scientific research, which allows them to explore data with JPSvis Online in a different way than before.

The fourth chapter is the main part of this work. It describes the implementation of all components in JPSvis Online in an exquisite detail and explains how all components work together to achieve the requirements discussed in the third chapter. The first section contains a description of JPSvis Online to show how users work with JPSvis Online. The second section deals with the implementation of all functional components with figures, tables and source code. Every detail of JPSvis is difficult to demonstrate in the thesis. This section tries to give a clear insight to the internal structure and logic of the software. The last section is the test. It follows the existing white box testing strategy to ensure that JPSvis Online works as expected.

Chapter 2

Scientific and Engineering Background

2.1 Analysis tools for pedestrian dynamics

As a complicated and continuous phenomenon, it is difficult to describe the macroscopic effects of pedestrian dynamics, such as collective effects and self-organization. From the microscopic view, pedestrians are three-dimensional objects, so a complete and accurate digitization of their movements and interactions is a challenge on an academic level. But without the scientific description, models for simulation and prediction of pedestrian dynamics cannot be developed.

However, it is always attractive and valuable to develop methods to analyze pedestrian dynamics. In the following sections there is an overview of the current state of knowledge and the available tools. These are relevant not only as a basis for clarifying the scientific requirements of the project, but also for the development of JPSvis Online.

2.1.1 Observables

In this section the observables that can be used to quantify pedestrian dynamics are presented.

At the macroscopic level, collective effects of pedestrian dynamics are often observed, e.g., jamming and oscillation, especially where space capacity is drastically reduced or inflow exceeds capacity. These phenomena are not caused by microscopic dynam-

ics, so the average speed of pedestrians cannot reflect their occurrence and varying process.

For this reason the flow J is introduced. The J of a pedestrian stream is the number of pedestrians who have crossed a defined area per unit of time[25]. If the measurement area is set to the location where a jamming is most likely to occur, such as at bottlenecks or doors, the process of jamming can be quantified.

The flow is defined as the rate of change of the pedestrian flow over the location, and there are two ways to calculate it. According to the definition of the flow J, in the time gaps $\Delta t_i = t_{i+1} - t_i$, two successive pedestrians i and $i+1$ passed through the fixed measurement location, the flow can be calculated as follows:

$$J = \frac{1}{\langle \Delta t \rangle}, \tag{1}$$

where the Δt may be identified:

$$\langle \Delta t \rangle = \frac{1}{N} \sum_{i+1}^{N} (t_{i+1} - t_i) = \frac{t_{N+1} - t_1}{N}. \tag{2}$$

The equation (1) and (2) shows that the flow J is a scalar quantity and an average value, not a momentary value at any given time.

Apart from the calculation from the definition, another approach to determine the flow is borrowed from the fluid dynamics [25]:

$$J = \rho v b = J_s b \tag{3}$$

where b is the width of the area crossed by pedestrians, ρ is the average density and v is the average speed. If the pedestrian flow is considered as a one-dimensional movement, i.e. the flow per unit of width, the specific flow is defined as:

$$J_s = \rho v. \tag{4}$$

The equation (3) introduces the relation between speed and density of pedestrians. In contrast to flow, velocity and density are the values at the microscopic level,

velocity describes the speed of change of the pedestrian's position and density is the mass per unit volume[11].

Depending on how the speed is calculated, there are differences between the average speed and the instantaneous speed. The average speed can be calculated as follows:

$$\bar{v} = \frac{\Delta x}{\Delta t} \tag{5}$$

which Δx is distance and $\bar{v}$ is the average velocity in the time period Δt.

At any particular time t, the instantaneous velocity can be calculated as:

$$v = \lim_{\Delta t \to 0} \frac{\Delta x}{\Delta t} = \frac{dx}{dt}. \tag{6}$$

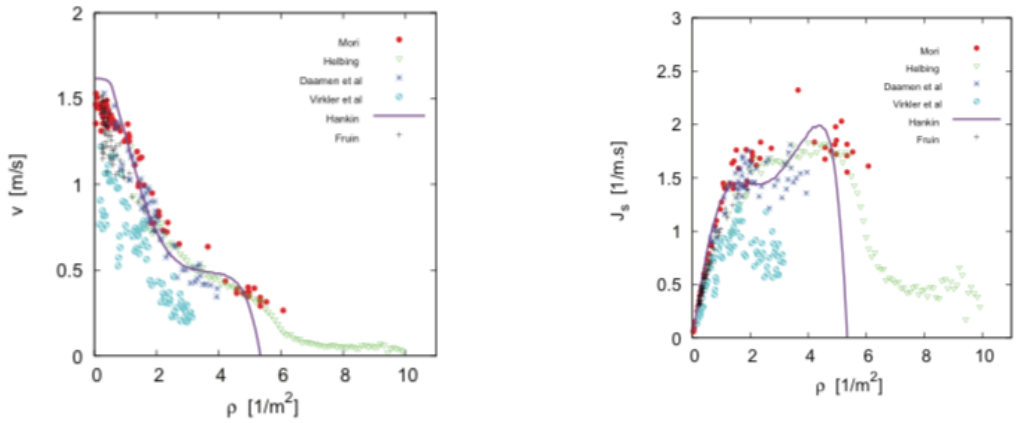
The physical definitions of them are clear, but there are problems in measuring and determining pedestrian dynamics, the difference in measuring methods for velocity and density will lead to a bias for the flow[25, 32], therefore the way of calculation for velocity and density must be decided and described in the source code of JuPedSim, see discussion in the section 3.3.

2.1.2 Fundamental diagram

In the section 2.1.1 the density ρ, the velocity v and the flow J are presented to describe the properties of the pedestrian flow. To provide inputs for engineering methods developed for the design of pedestrian facilities[10, 23] and a quantitative scale for models of pedestrian dynamics[5, 16, 19, 26], the fundamental diagram is used to represent the empirical relation between density ρ, velocity v and specific flow J_s[25].

In the pedestrian movement from commonly used manuals and empirical studies, there are two forms of the fundamental diagram that are commonly applied: $v(\rho)$ and $J_s(\rho)$. Figure 2.1 shows the fundamental diagrams of unidirectional pedestrian flow from various data sets of field studies.

Although the fundamental diagrams are influenced by the size and position of the measurement area, the type of flow and the flow ratio of the opposite pedestrian flow and other physiological, psychological and social aspects[32], general trends can be summarized in the fundamental diagrams.



(a) Density - Velocity

(b) Density - Specific flow

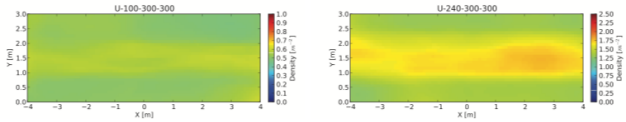
Figure 2.1: Fundamental diagrams of unidirectional pedestrian flow

Figure 2.1a shows the relationship between density and velocity. The movement of pedestrians requires a sufficient area for unrestricted pacing and sensory recognition of and response to potential obstacles[10], so that the speed of the pedestrian decreases with increasing density. As density increases, there is less free space for the individual pedestrian to move, limiting the walking speed required to avoid and adapt to slower moving pedestrians. This will always lead to a reduction in the speed of crowds of people.

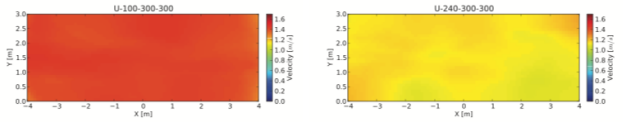
The relationship between density and flow J is also the most meaningful to observe the collective effects of pedestrian dynamics. By comparison with the equation (2) and (3) the specific flow J_s is independent of the width b in a given facility (e.g. corridors, stairs, doors), i.e. for a measuring range with different b, the fundamental diagrams $J(\rho)$ merge to a universal diagram for the specific flow J_s, it is also called specific flow concept, so that the fundamental diagram $J(\rho)$ from the hydrodynamic equation (3) is widely used. In the first phase, the specific flow increases with increasing density until the specific flow reaches the capacity of a facility, the jamming effect in the measurement area starts, pedestrians slow down and cause a decrease of J_s. As the density continues to increase, the congestion effect becomes more pronounced until the velocity drops to near 0.

2.1.3 Spatiotemporal profile

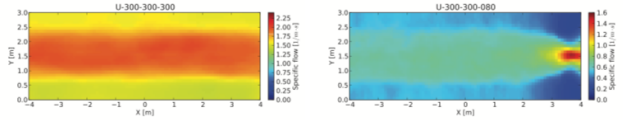
The spatiotemporal profiles of density ($\bar{\rho}(x, y)$), velocity ($\bar{v}(x, y)$) and the specific flow ($\bar{J}_s(x, y)$) present the spatial properties of motion for each run.



(a) Density profile



(b) Velocity profile



(c) Specific flow profile

Figure 2.2: Spatiotemporal profiles of unidirectional flow

Figure 2.2 is the density, velocity and specific flow profiles of unidirectional flow. From these profiles, the distribution of the observations is displayed in color, the influence of other factors, such as walls, can also be observed. This makes it even more important that the spatiotemporal profiles can show the transmission of the distribution during the simulation runs.

2.2 Visualization tools for pedestrian dynamics

With the analysis tools described in the section 2.1 the pedestrian dynamics can be quantified with parameters and plots. With the visualization of pedestrian simulation data, event organizers, safety and rescue authorities can know the scene in advance, navigate freely on site, and realistically depict the scene. this chapter describes existing visualization tools so that it is helpful to design the JPSvis Online.

JPSvis as a component of JuPedSim, an open source framework for simulation, analysis and visualization of pedestrian dynamics[15], can visualize the trajectory files (simulation and experiments) and geometry files generated by JPScore and JPSeditor as in the Figure 2.3.

The geometry file in xml format contains the architectural information of buildings, e.g. location of rooms and transitions. Usually the geometry files are drawn with JPSeditor, which is a CAD-like graphical user interface that allows JPSvis to design 2D and 3D scenes of the building. The trajectory files are the results of JPScore's

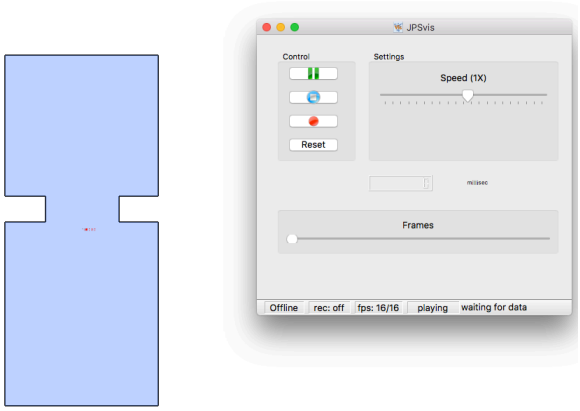


Figure 2.3: The User Interface of JPSvis

simulations, they contain the three-dimensional coordinates, the ratio of directions and other information of the pedestrians in each frame of a simulation run.

SumoViz is a web application for the visualization of simulation results of pedestrian flows from the Simulation of Urban Mobility (Sumo), which is an open source, highly portable, microscopic and continuous multi-modal traffic simulation package. SumoViz provides and displays the simulation results in two dimensions as in Figure 2.4. The presentation uses only HTML5 JavaScript and the Canvas API and is therefore accessible in all modern browsers[14].

Figure 2.4: The User Interface of SumoViz

SumoViz3D is intended to take up the approach of SumoViz, but extend the browser-based display by a three-dimensional view of pedestrian simulation data. In addition, values such as pedestrian speed or density can be read. In SumoViz3D it allows minor adjustments to the appearance of the scene and displays it as realistically as possible[4] as in the Figure 2.5.

Figure 2.5: The User Interface of SumoViz3D

2.3 Modern web technologies

The goal of JPSvis Online is to build a visualization and analysis tool based on web technologies and running in modern browsers. Therefore, this section introduces the relevant web technologies that are used in JPSvis Online.

All web-based applications are designed in the client-server model, which separates the client from the server. Each instance of client software can make a request to a server or application server[22]. The client-server model takes full advantage of the hardware environment at both ends to distribute tasks to the client and server, reducing the communication overhead of the system. An Internet browser acts as a client application and the Web server sends a request to retrieve an HTML page.

Hypertext Markup Language (HTML) is a standard markup language used to create Web sites and runs on and parsed in the browser, it is the standard publishing language of the World Wide Web, a network of information resources[8]. The latest version of HTML is HTML5. It has features such as a canvas for displaying images, animations, and 3D elements, support for multimedia, and tags for defining common document elements [18]. The specifications of HTML5 make it possible for JPSvis Online to display identical content in different browsers running on desktop and mobile devices.

The Canvas element is a new element introduced in HTML5 that allows dynamic display of 2D shapes and bitmap images and can display 3D shapes and images via WebGL[27]. It is like a curtain on which you can draw various diagrams, animations, etc. Without a canvas, drawings could only be created with Flash plug-ins, and the pages had to interact with JavaScript and Flash. With canvas, only the use of JavaScript is able to complete the drawing of 2D or 3D elements.

To edit HTML documents and create a dynamic website, the Document Object Model (DOM) is designed as programming APIs for HTML in browsers. It is a platform- and language-neutral interface that defines standards for accessing and editing HTML documents and allows programs and scripts to dynamically access and update the content, structure, and style of documents[31]. The DOM is divided into different models depending on the type of document, with the HTML DOM being the standard model for HTML documents.

HTML defines the standard for the World Wide Web in the data layer, the Hypertext Transfer Protocol (HTTP) defines the application protocol for distributed, collaborative, hypermedia information systems[20]. Version 1.1 of HTTP provides 8 methods to manipulate a given resource in different ways. For example, *Get* is the most common method to send a request for a given resource to the server. The *Post* method is used to send data to the specified resource for processing by the request server (e.g. submitting a form or uploading a file). The data is included in the request text.

JavaScript is an interpretive scripting language that can be embedded directly into HTML pages, but writing it as a separate js file makes it easier to separate structure and behavior. It is mainly used to add interaction to the HTML page, embed dynamic text into the HTML page, respond to browser events, read and write HTML elements, and perform other tasks[9]. With the development of the technology there are third-party libraries that use JavaScript as a core language to achieve three-dimensional modeling in the browser, network backend programming.

JavaScript is a so-called dynamic language in which type checking is performed at runtime, which ensures the flexibility of the language while creating pitfalls for large application development projects. TypeScript is a superset of JavaScript that supports the ECMAScript 6 standard[3]. TypeScript is designed for developing large applications that can be compiled into pure JavaScript, and the compiled JavaScript

can be executed on any browser. The front-end framework of JPSvis online is now developed by JavaScript and Typescript.

Chapter 3

Requirement and Solution Analysis

In this chapter the requirements for JPSvis Online are concertized on the basis of the goals discussed in the section 1.1, in meantime appropriate technical solutions are to be found to guide the implementation of JPSvis Online.

3.1 Overview

The web-based JPSvis Online should open in most modern browsers, such as Chrome, Safari and Firefox. It should work like a website, i.e. it starts after entering the URL in the browser. Users should first upload the geometry and trajectory files, then JPSvis Online can load the geometry directly to the website. On the geometry view page, users can set options to display pedestrian properties and play the simulation according to the trajectory file.

Regarding the analysis, users can upload different types of output files from JPSreport, JPSvis Online can generate corresponding diagrams, users have the possibility to download and save them on the local hard disk.

3.2 Software engineering

The visualization and analysis tool requires some key features in the field of software engineering to make it work for its purpose. The following sections 3.2.1 to 3.2.2 describe a variety of features that are fundamental to JPSvis Online. To implement

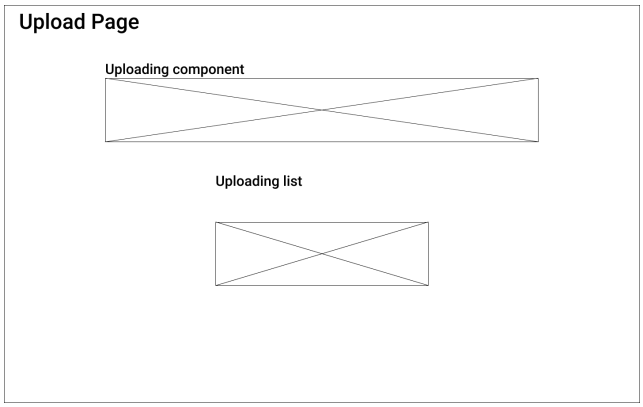


Figure 3.1: The User Interface draft - upload page

these features, several third-party Python and JavaScript libraries are introduced into the project at the same time.

3.2.1 User Interface

The user interface (UI) is basically divided into two pages: upload page and view page. In order to show the user a clear way to use JPSvis Online, the participants should first see what they see when they start uploading files. The Figure 3.1 shows a sketch of what the uploading page might look like. It should provide options for selecting files that are on the local hard drive through the upload component; if the upload is successful, the file name will be updated in the upload list.

The format of the geometry and trajectory files can only be the txt or xml format, and the format of the contents is also sensitive to parsing, so another feature of the upload page should be a filter for the files. If the format of the file and content is not valid, the upload page should return the error message to warn the users in the upload list.

Compared to the upload page there are more requirements on the view page. First, the geometry file should be displayed in the visualization component with 2D or 3D elements. Since the communication between server and client takes time and therefore it is not a good solution to render the 2D and 3D view on the server and then transfer it to the browser, it is better if the front-end framework can render animations locally.

Although JPSvis Online is not an application that focuses on performance, it is essential to ensure that a simulation in complex geometry can be smoothly visualized

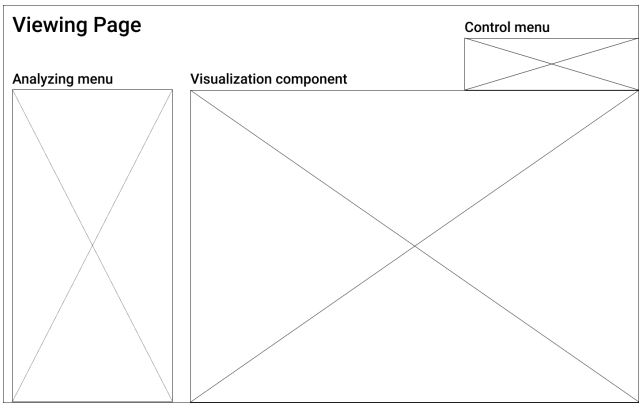


Figure 3.2: The User Interface draft - view page

in 3D view. The performance requirement is covered by the parameter frame rate of the 3D view.

The frame rate, or frames per second (FPS), determines how smooth the image of the animation is. The human eye perceives a sequence of images at about 15 frames per second [24] as motion, i.e. if the FPS of an animation is higher than 15, the human eye will consider it as animation, e.g. the FPS of movies are normally 24. However, 30 FPS is a better ground lever for 3D animations to ensure a stable view[28]. On the other hand, the upper limit of FPS depends on the monitor. Monitors are usually refreshed at 60 Hz to avoid flickering, so it is not necessary to drive the FPS of the visualization component higher than 60.

To switch the view between 3D and 2D, there is a control menu on the page. At the same time, the visualization component can use the control menu to adjust the viewing angle (in 3D view), show or hide pedestrian avatars and display other information. For the trajectory file, it should be automatically paring in the backend. When the user clicks on the playlist in the control menu, the avatars start playing the pedestrian trajectories.

With the exception of the visualization functions, the analysis options are offered in the analysis menu. Users should be able to select different diagrams depending on the output files uploaded to JPSvis Online. After selection, the diagrams should be generated in the backend and displayed directly in the analysis component so that users can view the simulation and analysis results simultaneously. When an analysis is complete, the diagrams can be saved to output files and users can download them.

To meet the above requirements, the front-end solution must be dynamic to update the components and interact with users in real time. Besides the usual HTML elements, JPSvis Online must also handle the interaction with the 3D scene for geometry.

3.2.2 Server

On the server side, it must provide front-end support throughout the process. For web development, the server means HTTP server (also web server). The most important task of the HTTP server is to receive an HTTP request from the browser and then send an HTTP response with files and content to the requester, as shown in the Figure 3.3, in other words, it provides a different programming interface (API) for the client side.

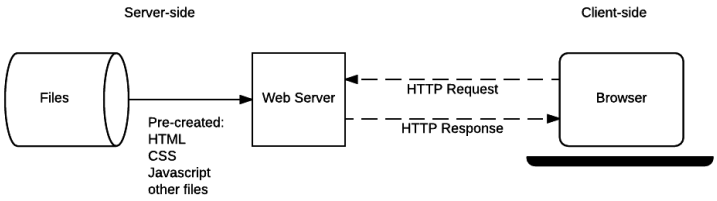


Figure 3.3: A basic web server

The first API is */upload*. On the upload page, the server should receive the geometry and trajectory files and return the result of the upload, with the result, the upload list knows which message to display. After uploads, the files should be read automatically and converted as JSON to Python. JSON refers to the JavaScript object representation, which is language independent and uses JavaScript syntax to describe data objects. The JSON format is syntactically identical to the code used to create JavaScript objects. Because of this similarity, JavaScript programs can use the built-in *eval*() function to create native JavaScript objects from JSON data without requiring a parser[21].

The second and third API are */geometry* and */trajectory*. When the upload is complete, JPSvis Online links to the view page by clicking. Before the view page opens, browsers send a request to load geometry and trajectory data in JSON from the server via these two APIs.

As described in subsection 3.2.1, the visualization component is built into the view page. When the user selects the diagram type in the analysis menu, the browser

sends the selected value to the server, according to the value the corresponding Python script for diagram generation is triggered. The analysis script is a part of the JPSReport source code, but in JPSReport they are used as a command line tool, after refactoring they can be used in JPSvis Online. For these diagrams the server must provide different APIs to answer requests individually.

Depending on the way the content is generated, there are two types of HTTP servers. A static web server for a static site returns the hard-coded content from the server when a specific resource is requested via an HTTP request. On the other hand, a dynamic web server responds to content generated by inserting data into placeholders in HTML templates, it can provide other data, or perform other operations as part of returning a response[17]. Since one of the goals of JPSvis Online is to provide user-friendly interaction with the visualization, a dynamic server is required for JPSvis Online.

However, JPSvis Online does not need to store user information, the simulation data is uploaded by the user login, so no database is required for the web server.

3.3 Scientific research

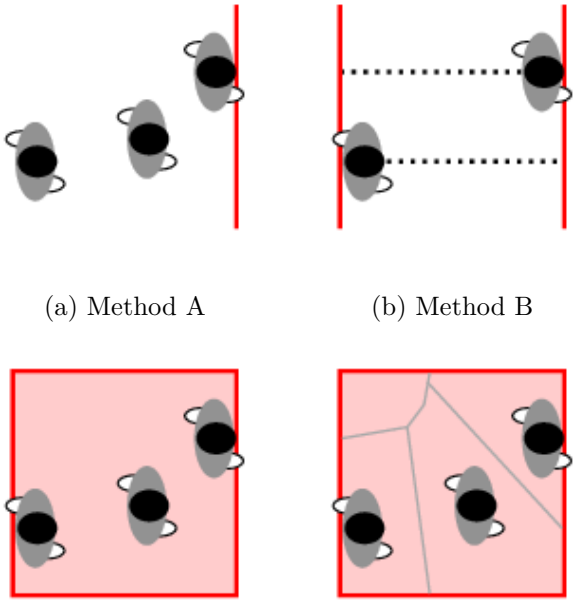
3.3.1 Analysis

This section discusses requirements for scientific analysis, such as methods for calculating density and velocity. These requirements will guide the development of JPSvis Online and can be met within acceptable ranges.

As discussed in the subsection 2.1.1, there are several problems regarding the way velocity and density are measured. In JPScore there are four methods to solve this problem, all of them can generate the result of the velocity or density and save the result in a text file in a different format.

The Figure 3.4 shows the setup for measuring pedestrians in research. Method A is used to calculate the mean value of flow and velocity over time with a reference line shown in the Figure 3.4a, in a given time period Δt the number of pedestrians who have crossed the reference line is determined by the equation (1) and (2), the flow over time $\langle J \rangle_{\delta t}$ and the time mean velocity $\langle v \rangle_{\delta t}$ can be calculated.

Using method A, the JPSreport outputs files with the name prefix $Flow\_NT\_traj\_$ as they are displayed in the Table 3.1a and $FDFlowVelocity\_traj\_$ in the Table 3.1b. The $Flow\_NT\_traj\_$ contains the time at which the pedestrian crosses



(a) Method A (b) Method B

(c) Method C (d) Method D

Figure 3.4: Methods for calculating density and velocity

the reference line and the cumulated number of pedestrians. This file can be used to create an N-t diagram in which this time is displayed as x-axes and the cumulative number of pedestrians as y-axes.

Time [s]	Cumulative pedestrians
0.00	0
0.88	0
0.94	1
1.22	2
...	...

(a) Flow_NT_traj_

Flow rate(1/s)	Mean velocity(m/s)
8.107	0.943
7.243	0.425
7.368	0.262
...	...
2.989	0.000

(b) FDFlowVelocity_traj_

Table 3.1: Output files with Method A

In contrast to method A, methods B, C and D allows the simultaneous measurement of velocity and density with different setups. With method B, a segment Δx is defined as the measuring range in a corridor as in Figure 3.4b. For each pedestrian,

Person Index	$Density\_i(m^{-2})$)	$Velocity\_i(m/s)$
1	0.678161	3.448276
2	1.097884	3.174603
3	0.635593	3.389831
4	1.119444	3.333333
...	...	

Table 3.2: Output files with Method B

the times of entry and exit from the area are recorded so that the speed $\langle v \rangle_i$ is calculated as:

$$\langle v \rangle_i = \frac{\Delta x}{t_{out} - t_{in}}, \tag{7}$$

and the density of person i is:

$$\langle \rho \rangle_i = \frac{1}{t_{out} - t_{in}} \cdot \int_{t_{in}}^{t_{out}} \frac{N'(t)}{b_{cor} \cdot \Delta x} dt, \tag{8}$$

where b_{cor} is the width of the measurement area while $N\ (t)$ is the number of person in this area at a time t.

Using method B, JPSreport outputs a file with the prefix $FDTinTout\_traj\_$, as in Table 3.2, which contains the density and speed of each pedestrian in a steady state. However, for a fundamental diagram JPSreport needs the density and velocity data in time series, so it is not necessary to implement diagrams from output files based on method B in JPSvis Online.

Unlike Method A and B, the measuring range of Method C is set as a rectangle as in Figure 3.4c to calculate the density with the classical method:

$$\langle \rho \rangle_{\Delta x} = \frac{N}{b_{cor} \cdot \Delta x}, \tag{9}$$

where b_{cor} stands for the width of the rectangle and Δx for the depth. The method C creates the file $rho\_v\_Classic\_traj\_$ for the specific measurement area i like in the Table 3.3.

Frame	$density(m^{-2}))$	$velocity(m/s)$
00103	0.000	0.000
...	...	...
00150	0.167	3.476
00151	0.167	3.427
...	...	

Table 3.3: Output files with Method C

So this output file is necessary to implement because it contains the density and speed in the range i during the whole process. With the columns $Frame$ and $Density$ it can be used to create a $\rho - T$ diagram to show changes in density during the whole simulation process, with $Frame$ and $Velocity$ for the $V - T$ diagram, with $Density$ and $Velocity$ for $v(\rho)$, and based on the Equation 3 the $J_s(\rho)$ diagram can be created.

Method D also uses a rectangular area to measure density and velocity, but unlike method C, method D uses the Voronoi method to calculate density. The Voronoi cell A_i in the Figure 3.4d, the density of space ρ_{xy} is defined as:

$$\rho_{xy} = \frac{1}{A_i}, \tag{10}$$

so the Voronoi density for the measurement area can be calculated as:

$$\langle \rho \rangle = \frac{\iint \rho_{xy} dx dy}{b_{cor} \cdot \Delta x}. \tag{11}$$

With method D the output file $rho\_v\_Voronoi\_$ is generated like the output file of method C, so that this file is also a possible data source for the presentation of the fundamental diagram $\rho - T$, $V - T$, $v(\rho)$ and $J_s(\rho)$, which should be implemented in JPSvis Online. Apart from the fundamental diagrams, method D can also generate the files $IFD\_I\_traj\_.dat$, which contain the profile data for density and velocity, therefore profiles of density and velocity should be plotted.

In summary, method A will provide the data for the NT-diagram, method C and D will develop the basic diagrams $\rho - T$, $V - T$, $v(\rho)$ and $J_s(\rho)$. As density and velocity profiles they should be created with the output files of method D.

In JPSvis Online the output files are parsed with Python script, the Python library *Mathplotlib* is used to plot the fundamental diagram. *Mathplotlib* is written in Python and works as a 2D graphics package. It is widely used in application development, interactive scripting, and publication-quality image generation across user interfaces and operating systems[13].

3.3.2 Visualization

One of the goals of JPSvis Online is the visualization of pedestrians and their locations in 2D and 3D buildings, therefore JPSvis Online should be able to analyze the files containing location information of pedestrians and buildings.

The spatial information of buildings is usually stored in the geometry XML file as in the Listing 3.1.

```
<geometry>
    <rooms>
        <room>
            <subroom>
                <!--walls of the subroom-->
                <obstacle>
                    <!--obstacle inside the subroom-->
                </obstacle>
            </subroom>
        </room>
    </rooms>
    <transitions>
        <!--doors between two rooms or a room and the outside-->
    </transitions>
</geometry>
```

Listing 3.1: The structure of geometry xml

The root tag of the geometry file is always *geometry*, in the next level there are the tags *rooms* and *transitions*. Rooms contains at least one room, which means a part of the building, it can be room, corridor, lobby, platform or staircase and so on. In a room there can be only one subroom, depending on the class of the subroom, there is different information contained in the *subroom* tag. For example, for stairs there are *up* and *down* tags to indicate the direction of the stairs, and for platform the label of polygon is defined as *track*.

The XML geometry file is parsed using the official Python library *xml.etree*. The *xml.etree.ElementTree* module in the library implements a simple and efficient

API for parsing and creating XML data[30]. After parsing, texts in the XML file are restructured as JSON format to be transferred between browser and server using HTTP methods,

It is worth mentioning that the XML file describes the geometry in 2D space, so the geometry information should be extended as a three-dimensional element in JPSvis Online. After the simulation in JPScore the trajectory txt-file of pedestrians can be generated, it describes all information pedestrians in each frame during the simulation.

```
#ID FR  X   Y   Z   A   B   ANGLE   COLOR
1 0 3.30  3.33  0.00  0.18  0.25  -90.00  0
2 0 4.50  4.44  0.00  0.18  0.25  -90.00  0
3 0 3.60  3.70  0.00  0.18  0.25  180.00  0
4 0 3.60  4.07  0.00  0.18  0.25  180.00  0
5 0 4.50  4.07  0.00  0.18  0.25  -90.00  0
6 0 4.20  3.33  0.00  0.18  0.25  -90.00  0
```

Listing 3.2: The structure of trajectory file

As shown in the Listing 3.2, each line in the file shows information about a pedestrian in a frame. X, Y, Z are the coordinates in meters, A, B are the half-axes of the ellipse, angle means the direction of the pedestrian and color stands for the speed.

With the standard function *open* the simple txt file can be read into memory, the challenge is to convert this information into another format, with this format the file can be transferred and read with JavaScript. To do this, a script in Python must be available on the server side to translate the data.

Chapter 4

JPSvis Online

This chapter aims to describe the implementation of JPSvis Online and its workflow in detail. It lists individual component and how components cooperate with each other. Thereby an overview of the whole program is given at first, the implementation of every module will be described respectively.

4.1 Architecture

The motivation of JPSvis Online is to enable the visualization of pedestrian trajectories and other additional exploratory information in the field of pedestrian dynamics. As described in the section 2.2, JPSvis already almost achieves the goals mentioned in the section 1.1, except for the feature: cross-platform.

In a short time, however, it is not realistic that the construction of JPSvis Online is based on the original JPSvis code. JPSvis is mainly programmed in C++ using the Qt framework for the user interface and the Visualization Toolkit (VTK) for 3D graphic visualization. Although the Qt framework and VTK provide their own cross-platform solution for Windows, MacOS and Linux, they are implemented as a C++ toolkit and require users to create applications by combining different objects in C++ into one application[12, 7], therefore they cannot work with JavaScript. There are also no official JavaScript-based libraries of Qt and VTK, therefore the architecture and source code of JPSvis cannot be a reference for JPSvis Online.

As discussed in the section 3.2, the web application developed with the client-server model consists of two main components: client and server, these two components are implemented by a number of functional modules, as shown in the Figure 4.1.

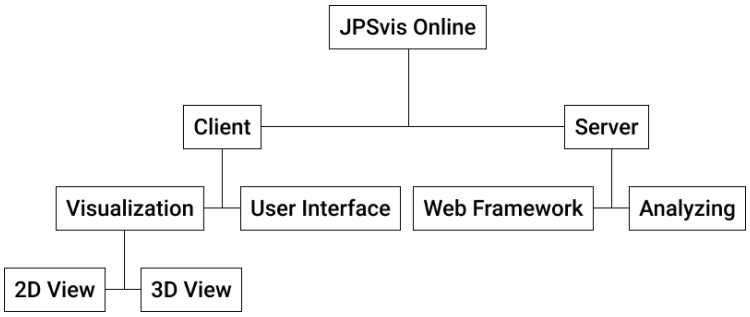


Figure 4.1: The Architecture of JPSvis Online

The most basic module on the client side is the UI, it contains all the essential graphical widgets for the user in the browser, such as buttons, text, interactive windows, menus and tabs. Although all widgets can be designed and constructed using pure HTML elements and CSS styles, this is a time-consuming process, which is also not robust due to the complexity of software development. Therefore, it is necessary to introduce the React library for user interface design to reduce the complexity of the user interface. Compared to the crowded category of UI libraries such as jQuery, Angular, Vue, Meteor and others, React is a declarative, efficient and flexible method of user interface creation. The complex UI consists of small and isolated pieces of code called ”components”. React was released as open source and described as ”V” in the MVC model. In other words, components by React acted as the view layer or user interface for websites[2].

However, React does not include tools for manipulating the *canvas* element to render 2D and 3D geometry from the geometry file. Although most modern browsers support WebGL APIs to draw content in the *canvas* element, WebGL is a very low-level system that can only draw points, lines and triangles. Trying to do something practical with WebGL usually requires a lot of code. For this reason the three.js are used to render geometry and trajectory files in 3D view. The three.js is a cross-browser JavaScript library programming interface used to create and display animated 3D computer graphics. It can help us work efficiently with scenes, lighting, shadows, materials, textures, spatial arithmetic, pretty much everything you need to do yourself via WebGL[6].

By adjusting the camera in the scene created by the three .js, the geometry and trajectories can be viewed in 2D, but this way is not interactive for the user. For a better representation of geometry and trajectories in 2D view the PixiJS is intro-

duced. PixiJS is an extremely fast 2D rendering engine that helps the browser to display, animate and manage interactive graphics in the *canvas* element[29].

On the server side, as discussed in section 3.2, Python is selected for server-side encoding. The main tasks on the server side are to work directly with HTTP requests and responses via the HTTP protocol, to route requests to the appropriate handler, to access the data in the request, and to render the data into an HTML template. To achieve these goals more easily, the Python library aiohttp is integrated, providing tools for building an asynchronous HTTP server. To analyze the requests, the scripts for generating diagrams will be included in the functions of the request handler.

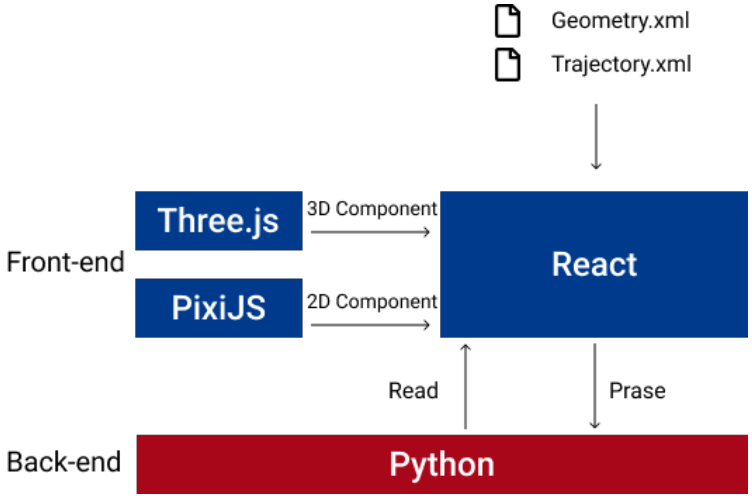


Figure 4.2: The tech stack of JPSvis Online

The tech stack of the entire program is shown in the Figure 4.2 and the sequence diagram is shown in the Figure 4.3. To keep it clear, the sequence diagram contains only the abstract layers. When JPSvis Online is opened, the React application starts generating the website and returns the HTML file to the browser. Users interact with the React components, for example by uploading geometry and trajectory files. The files are sent to the server to be parsed in JSON format and returned when the view page is requested. On the view page, the three.js will render the geometry in 3D and PixiJS will change it to 2D. Users can also interact with 3D or 2D content on the page. At the end the users set the diagram type, the server returns the diagram and displays it on the view page.

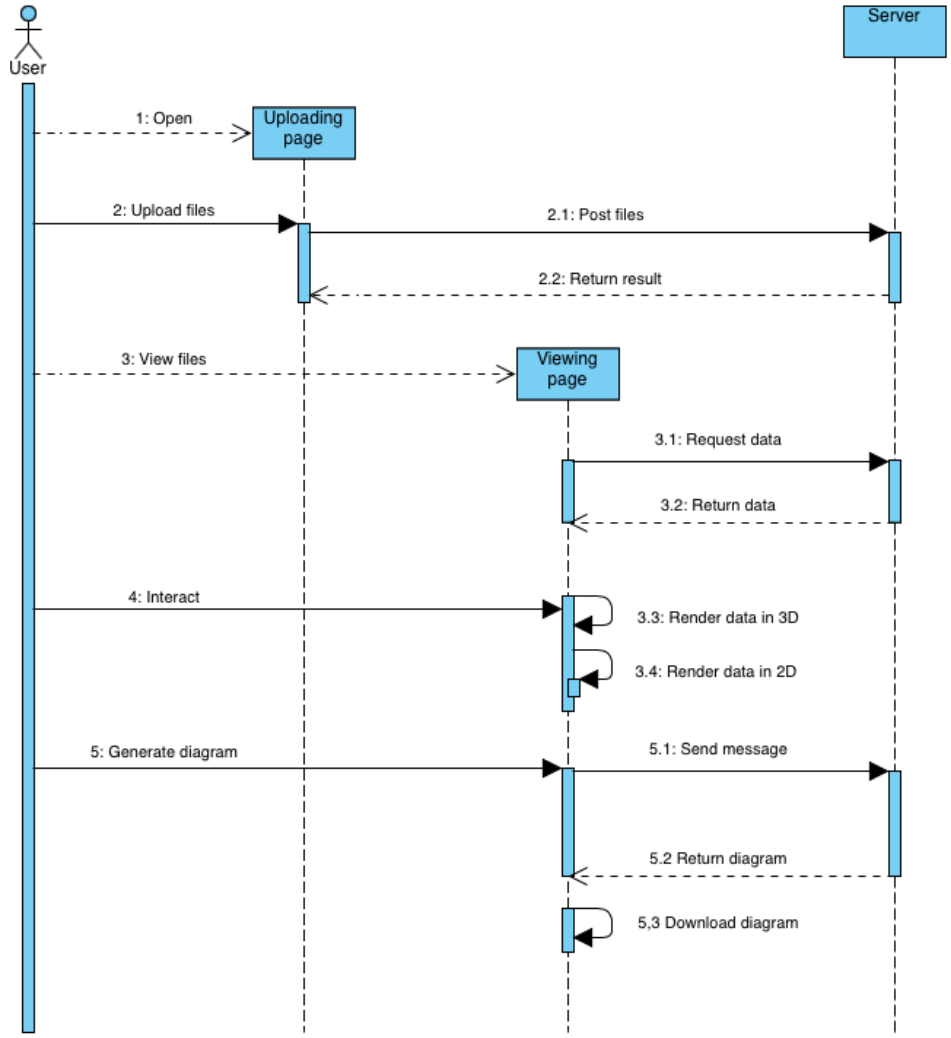


Figure 4.3: The sequence diagram of JPSvis Online

4.2 Implementation

The following sections describe the implementation in source code of the features discussed in the section 3.2.

4.2.1 HTTP server

As discussed in the subsection 3.2.2, the task of the HTTP server is to provide responses to requests from the website, so the server is set up before JPSvis Online is opened.

```python
def setup_server():
    app = web.Application()

    cors = aiohttp_cors.setup(app, defaults={
        "*": aiohttp_cors.ResourceOptions(
            allow_credentials=True,
            expose_headers="*",
            allow_headers="*",
            allow_methods="*",
            max_age=3600,
        )
    })
    resource = cors.add(app.router.add_resource("/upload"))
    cors.add(resource.add_route("POST", post_file))

    # Routers
    app.router.add_get("/", index)
    app.router.add_get("/ViewPage", index)
    app.router.add_get("/geometry", get_geometry)
    app.router.add_get("/trajectory", get_trajectory)
    app.router.add_get("/N_t", get_Nt)
    app.router.add_get("/Profiles_Density", get_profile_density)
    app.router.add_get("/Profiles_Velocity", get_profile_velocity)
    app.router.add_get("/Density_Time", get_density_frame)
    app.router.add_get("/Velocity_Time", get_velocity_frame)
    app.router.add_get("/Density_Velocity", get_density_velocity)
    app.router.add_get("/Density_Flow", get_density_J)

    app.router.add_static('/', path=str(PROJ_ROOT / 'static'))

    return app
```

Listing 4.1: Set-up web application

The server for JPSvis Online is based on the aiohttp package, the first step is to create an application instance (a synonym for web server) from the function *web.Application* as shown in the Listing 4.1. The *web.Application* function returns an application that is actually a dict-like object. The main task of the application is to handle HTTP requests, i.e. an application contains a router instance and a list of callbacks that are called by HTTP requests. These HTTP requests are mapped by event handlers using the method *add_get*, e.g. if the router */Viewpage* is requested using the GET method, the function *index* is called. The second task of

the application is to map static content like index files, images, JavaScript and CSS files to the root folder using *add_static*.

The last task of the web application is the implementation of Cross-Origin Resource Sharing (CORS). To ensure browser security, all modern browsers follow a same-origin policy. If a web script in one page wants to access data on another page, the two pages must be from the same origin, i.e. they have the same URI, host name and port. In JPSvis Online the uploaded files are stored in */upload*, if the */ViewPage* wants to read the data to visualize the geometry, the operation is rejected by the browser due to the equal origin policy. To deal with this limitation, the CORS mechanism has been included in the W3C standard. The key to implementing CORS communication is the server. As long as the server implements the CORS interface, it can communicate across sources.

The function *aiohttp_cors* is used to set up CORS for the JPSvis online server, the function *aiohttp_cors.setup* configures the application and activates CORS on resources and routes that need to be exposed, i.e. */upload* in the project, then the resource on the */upload* page is available to all sources with non-permitted passing of credentials. With the exception of the resource, the *POST* method and its handler *post_file* is also defined.

The application is configured, the next step is to run the application. First, a runner is created for the application, which serves on specific ports. Then the corroutine *runner.setup* initializes the application which can be served on a specific TCP socket. At the end the *start* function starts handling a site.

As a web application, JPSvis Online is most likely visited by multiple users from multiple IPs, which means that the server must take into account a high concurrency the design of the server to ensure that the system can handle many requests simultaneously and in parallel. There are a number of approaches to deal with high concurrency, the most common solution in Python is to use an asyncio package, it implements asynchronous IO in Python to handle the challenge of high concurrency. The programming model of asyncio is an event loop. Through *asyncio.get_event_loop* a reference to the event loop is retrieved directly from the asyncio module and the concatenation to be executed is then thrown into the event loop to execute it, thus implementing asynchronous IO. The web server runs continuously until it is stopped by an external command.

```
1
2  async def init():
3      app = setup_server()
4      runner = web.AppRunner(app)
5      await runner.setup()
6      site = web.TCPSite(runner, '0.0.0.0', 8080)
7      await site.start()
8
9      return app
10
11 if __name__ == '__main__':
12     loop = asyncio.get_event_loop()
13     loop.run_until_complete(init())
14
15     try:
16         loop.run_forever()
17     except KeyboardInterrupt:
18         pass
```

Listing 4.2: Run web application

When the server is running, HTTP requests receive the correct and timely response. As can be seen from the Listing 4.1 the server will execute *index* when JPSvis Online is opened. The function *index* opens and reads the file *index.html*, which is stored in the root folder as a static file, and sends the content of *index.html* as response to the browser. When the browser receives the response and starts to parse it, the script in the file is triggered, the script flow is described in subsection 4.2.2.

Another import handler is $post\_file$ for uploading files. As can be seen from the Figure 4.4, after receiving the $POST$ request from the browser, since files in the HTTP request are converted to text, the first step is to check if the uploaded files are readable; if they are, they are saved as a file again. The name of the file is also stored in the header of the request; to make it easier to use later, these files are renamed according to the format in which they are named. For example, whatever the original name of the geometry file is, it is renamed to *geometry.xml*. If the renaming is successful, the server will return the success code to the browser, the website will indicate that the upload has been completed successfully, the uploaded files will be stored in the server folder where the *server.py* file is located.

When the view page is opened, the browser sends the requests $/geometry$ and $/trajectory$ to retrieve data from the files (see details in subsection 4.2.3 and 4.2.4. The format of the geometry files is XML, it is easy to use the *xmltodict* package to

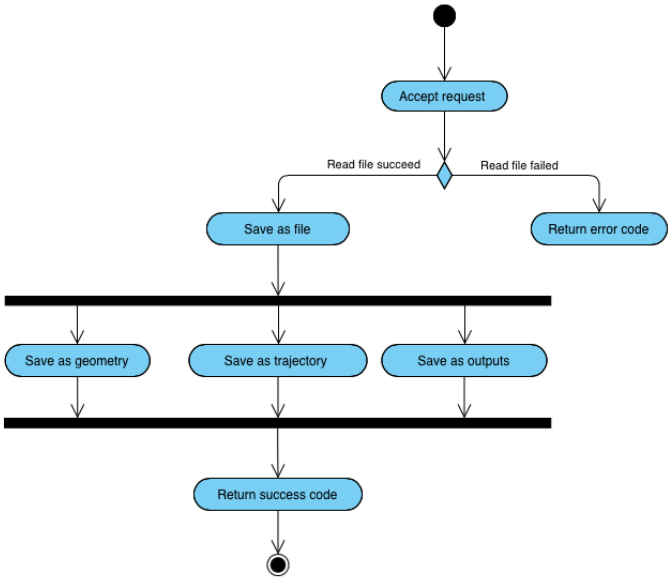


Figure 4.4: The activity diagram of $post\_file$

convert it to a dictionary, but the format of the trajectory files is *txt*, so it must be read by script and its contents must be saved to a dictionary. However, the HTTP protocol does not allow to transfer data as Python dictionary, it must be serialized with $json.dumps$, it serializes the dictionary as a JSON formatted stream to the JSON file. At the end the geometry and trajectory JSON files are sent to the client side.

Besides the handlers for uploading files and retrieving data, the other APIs are for the generated diagram. These scripts are described in detail in subsection 4.2.5.

4.2.2 User Interface

The implementation of UI follows the requirements in the subsection 3.2.1. In the web pages or applications, all function modules are embedded in HTML tags to present content or provide widgets, therefore the HTML tag tree implicitly shows the structure of JPSvis Online.

The Figure 4.5 shows the HTML tag tree of the file (see Listing 4.3). $<!DOCTYPE html>$ declares this file as an HTML5 document, and the element $<html>$ is the root element of an HTML page. For each HTML file, it consists of the tags $<head>$ and $<body>$, the $<head>$ element contains the metadata

for the document, and below it there are three tags. $< meta >$ defines the encoding format of the web page as utf-8, the $< title >$ element describes the title of the document, and the $< link >$ element refers to the reference of the CSS file.

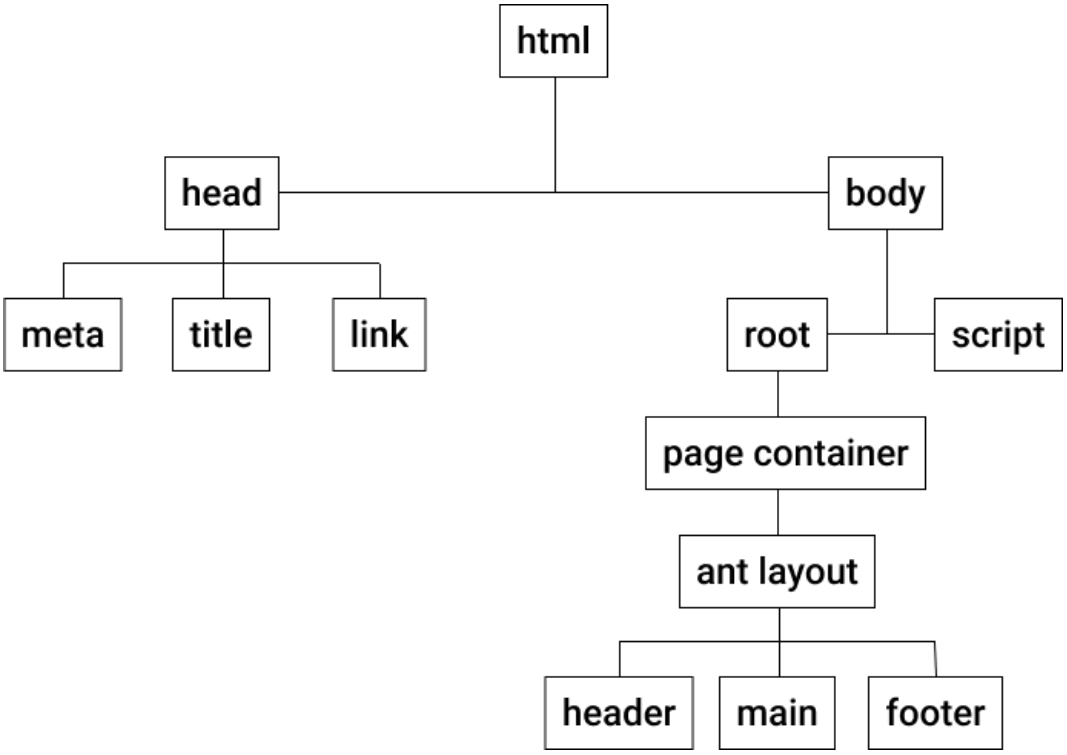


Figure 4.5: The tag tree of the uploading page

The element $< body >$ contains the visible content on the webpage. Unlike other UI libraries, React does not manipulate the HTML file directly, but appends elements in a root tag in the HTML after rendering in a virtual DOM. With this feature it is possible to create JPSvis Online as a one-page web application, only the *index.html* has to be uploaded to the server, the content for upload and view pages is rendered and managed by the root react component.

In the file *index.html* the $< script >$ tag determines which script will execute a React application. After rendering in the React script, the content is appended under the $< root >$ tag. Once the content of React has been modified or users have interacted with it, React renders a virtual DOM tree and compares it to the DOM tree in the *index.html* file. If there are differences between the trees, React will update the content in the page under the $< root >$ tag.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>JPSvis Online</title>
    <link rel="stylesheet" href="/index.css" />
</head>
<body>
<div id="root" class="layout-parent"></div>

<script src="/index.bundle.js"></script>
</body>
</html>
```

Listing 4.3: Index HTML

When JPSvis Online is opened, the *ReactDOM.render* function starts to render a React element $< App/ >$ into the DOM as in the Listing 4.4. The statement $document.getElementById('root')$ defines that the React element is mounted into the provided container *root* in the file *index.html*.

```
  const render = () => {
    ReactDOM.render(
      <App />,
      document.getElementById('root')
    )
  }
  render()
```

Listing 4.4: Render root into the DOM

After mounting in the file *index.html* the function *App*() is called as shown in the Listing 4.5. As discussed above, JPSvis Online is a one-page application and the upload page and the view page should not merge into one page, so there must be a mechanism to handle the route for changing pages. The $react - router$ package provides the basic routing functionality for React, two routes for the upload page and the view page are set under the *Router* component, and the upload page as the default page. By calling the $/ViewPage$ link, the browser is linked to the view page with the same *index.html* in the server, saving time managing HTML and other static files.

```
const App = () => (
  <Router>
    <div id='page-container' className='pages-container'>
      <Switch>
        <Route exact={true} path='/' component={UploadPage}/>
        <Route path='/ViewPage' component={ViewPage}/>
      </Switch>
    </div>
  </Router>
)
```

Listing 4.5: App component

The upload page are displayed in the Figure 4.7, it is built from three components of Ant Design: *Header*, *Content* and *Footer* with a simple "header-content-footer" layout, the layout of the whole page is stable, it is not influenced by the view area. In general the main navigation bar (see no.1 in Figure 4.7a) is placed in *Header* at the top of the page and contains the logo, the first level navigation (starting with the button, link button to the official JuPedSim website), from left to right in it. The *Content* contains the functional components. The *Steps* is a navigation bar (see no.2 in Figure 4.7a) which guides the user through the steps of uploading.

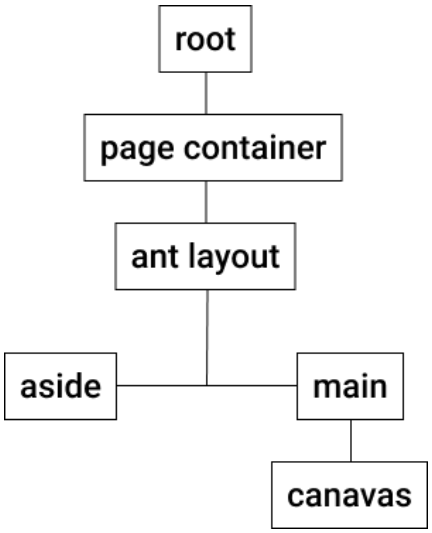


Figure 4.6: The tag tree of the viewing page

In the first step the user should upload the geometry file (see Figure 4.7a), then in the second step upload the trajectory file (see Figure 4.7b). With the upload widget under steps, users can drag files to the specific area (see no.3 in Figure 4.7a) to upload them. Alternatively, users can upload by selection. The buttons (see no.4 in

Figure 4.7a) under the upload widget should trigger the appropriate business logic; after uploading, click the *StartVisualization* button (see no.1 in Figure 4.7b) to go to the view page.

Unlike the upload page, which consists of all React components, the view page must contain React components, three.js component and PixiJS component, accordingly the HTML tag tree of the view page differs under *antlayout* tag. As shown in the Figure 4.6, the view page does not follow the "header-content-footer" layout, but the two-column layout.

The upper part of sider (see no.1 in Figure 4.8) is used to upload the output file of JPSReport for analysis, then users can select the type of diagrams using the cascade selection box (see no.2 in Figure 4.8). According to the requirements of subsection 3.3.1, the following diagrams are offered for selection (see Figure 4.9a)

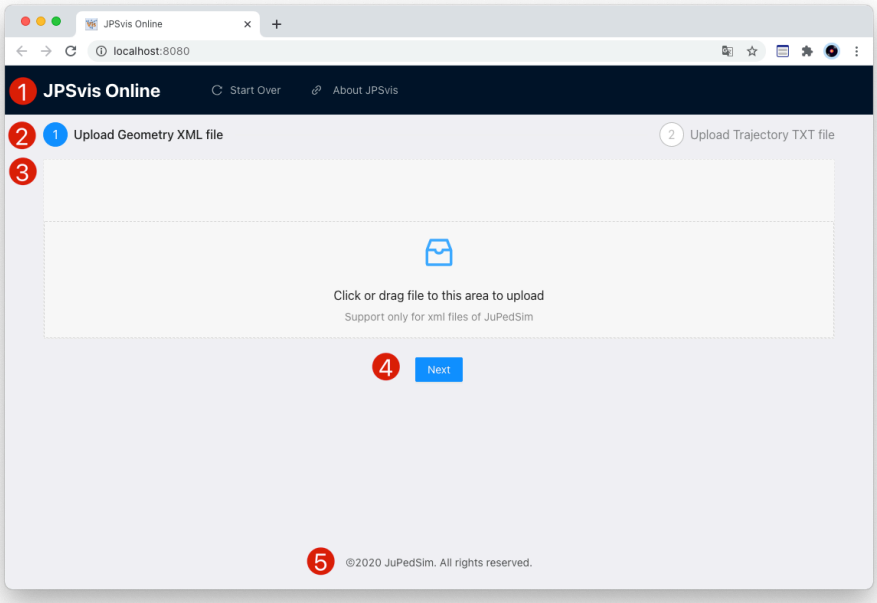
1. N-t Diagram
2. Density - Frame Diagram
3. Velocity - Frame Diagram
4. Density - Velocity Diagram
5. Density - Specific Flow Diagram
6. Density profile
7. Velocity profile

When the diagram type is selected and the corresponding output files are uploaded, click "Start Plot", the generated diagram will be displayed in the popup window in the Figure 4.9b. Users can download the diagram directly from this window. For more details on how diagrams are generated, see the subsection 4.2.5.

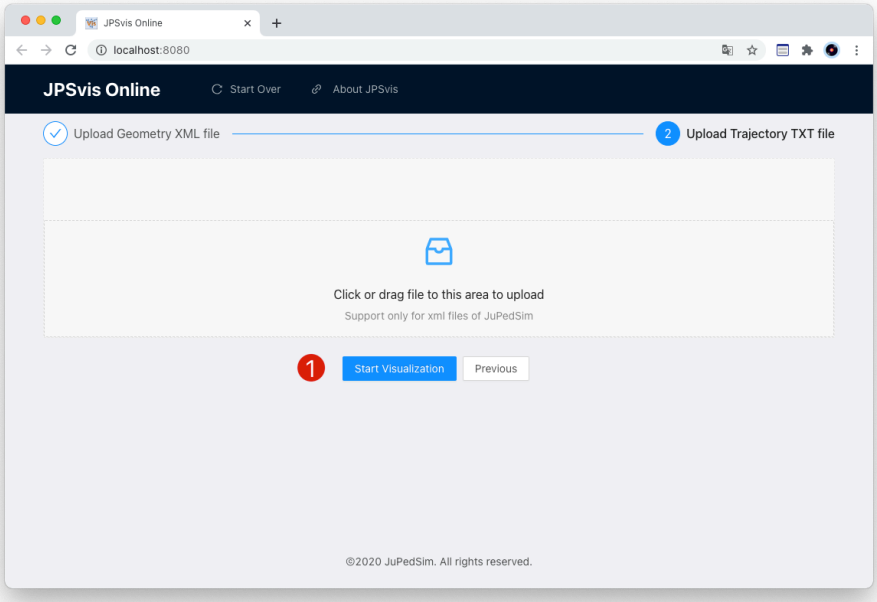
Besides the *aside* tag for sider on the view page, there is a *main* tag as a container for *canavas* for 3D and 2D view (see no.3 in Figure 4.8). The three.js and PixiJS use the same *canavas* to render content on the view page.

4.2.3 3D visualization

As discussed above, React cannot provide components for rendering geometry and trajectory files, so the three.js is inserted into the *canavas* tag after the view page is mounted.



(a) The uploading page for geometry files



(b) The uploading page for trajectory files

Figure 4.7: Screenshots of the uploading page

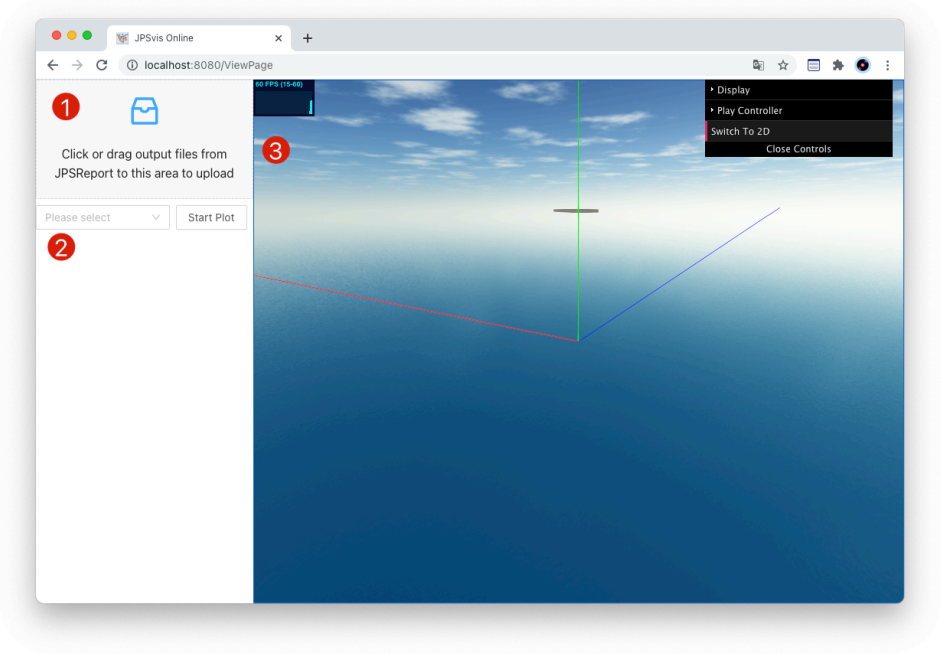
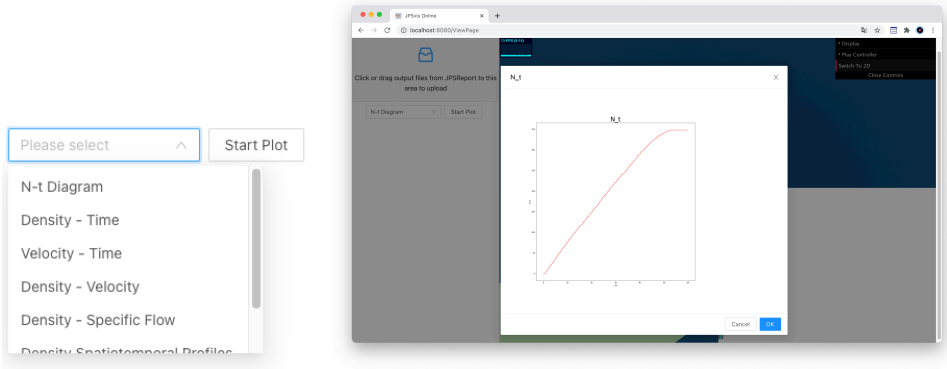


Figure 4.8: The screenshot of the viewing page



(a) The diagram menu (b) The popup window for showing diagram

Figure 4.9: Screenshots of the diagram window

```
componentDidMount(){
  (async () => {
      const initResources = await init();
      this.jps3D = new JPS3D(initResources.geometryRootEl,
          initResources);
    }
  )();
}
```

Listing 4.6: Instantiating for 3D view

The *componentDidMount*() (see the Listing 4.6) is invoked immediately after the viewing page component is mounted into the HTML tree. The instantiating for *JPS3D* class need to load geometry data from a remote endpoint, so there is a immediately-invoked function expression with *async* keyword. The Async function contains a await expressions for *init*().

Await expressions pauses the execution of the asynchronous function $fetchJson$ and waits for Promise to execute, then continues to execute the asynchronous function and returns the result. The return value after parsing is treated as the return value of the await expression in the promise. This is the reason why the async function is needed here, *init*() sends requests to server to fetch the data of geometry and trajectory, and save them into object *initResources*.

```
export default async function init (): Promise<InitResources> {
  const geoData = await fetchJson<GeoFile>('geometry');
  const traData = await fetchJson<TraFile>('trajectory')

  return{
    geometryData: geoData,
    geometryRootEl: getOrThrow('canvas'),
    trajectoryData: traData
  }
}
```

Listing 4.7: The initiating of data

The class *JPS3D* renders the 3D view using three.js as in the Figure 4.10. The geometry and trajectory data are set as parameters for the constructor of the class JPS3D and stored as *init* property, the *geometryRootEl* defines the HTML tag of *index.html* that three.js should connect , namely *canvas*.

To start displaying anything with three.js, a scene, a camera and a renderer should be initiated. The scene is the place where any objects can be added. The camera decides how the scene is displayed, in three.js there are four parameters that need to be set to adjust the camera. The first attribute is the field of view (FOV), it is the range of scenes that can be seen on the screen at a given time. The second is the aspect ratio, which is used to scale the screen so that it can be adapted to different content. The next two attributes are the near and far clipping planes. This means that objects that are further away than the far or closer than the near are not rendered with them the pressure on the rendering in the backend can be controlled below the limit. At the end is the renderer, in the section 4.1 the difference between WebGL and three.js is explained, but actually three.js uses *WebGLRenderer* by default to communicate with the JavaScript core in browsers.

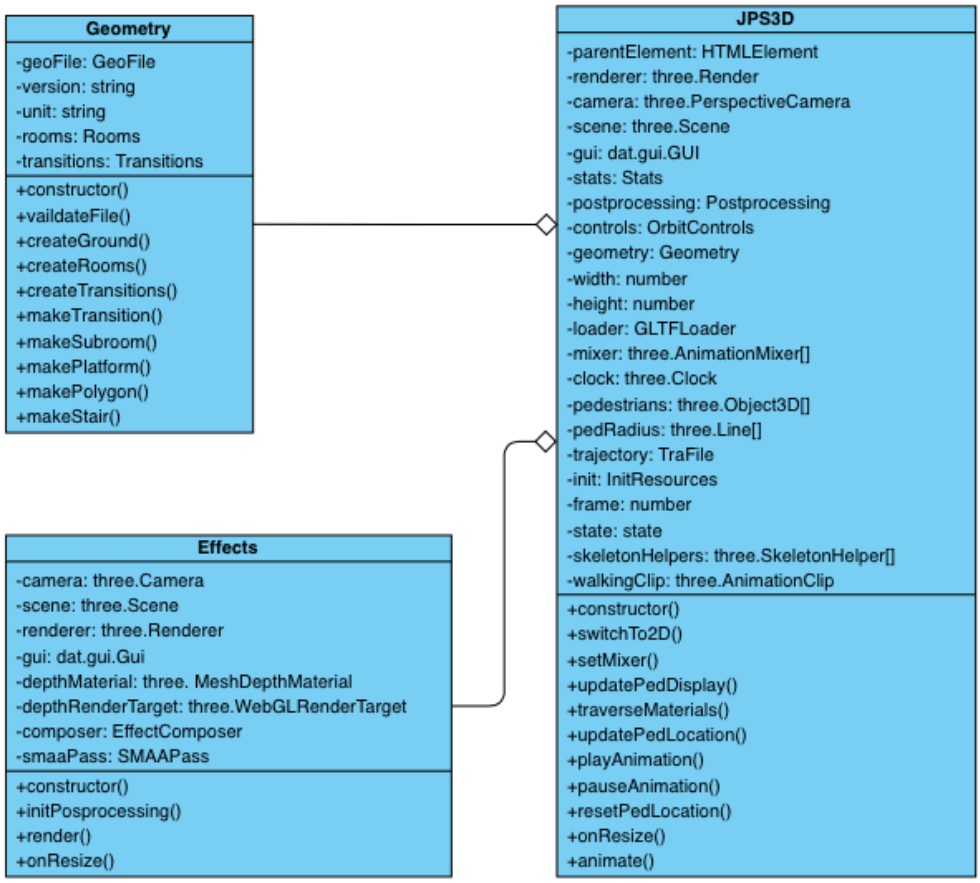


Figure 4.10: Class diagram for 3D view

After the initialization of three.js, a black scene is shown at the position where *canavas* is, next comes the light and the control for the scene. Light is a very

complex physical phenomenon in the real world, to simulate it in the 3D view, three.js provides different types of light. First the *AmbientLight* is set to simulate ambient light, it illuminates globally and evenly all objects in the scene. The *AmbientLight* can improve the luminance of the scene, but it is not directional, so the *DirectionalLight* class is added to simulate sunlight. The sun is far enough away from us so that its position can be considered infinite, and all rays emitted by it are parallel, so the *DirectionalLight* emits from a certain direction as if it were infinitely far away, and all rays of light produced by it are parallel. With *AmbientLight* and *DirectionalLight* the room is illuminated enough but the contrast between light and shadow is too strong, so the class $HemisphereLight$ is needed, it is placed directly above the scene and the color fades from the sky to the ground.

The mouse is the main input device for the users and the 3D view for navigation. In the three.js the $OrbitControls$ class allows the camera to circle around a target. For setting the controls $maxDistance$, $minDistance$ and $maxPolarAngle$ must be defined. $maxDistance$ and $minDistance$ control how far you can extend and retract, both default settings are infinite. $maxPolarAngle$ determines how far you can circle vertically.

For now, an empty 3D scene is created, the next steps are adding geometry and pedestrian trajectory based on the uploaded data. The geometry is created by the class *Geometry* (see Figure 4.10), which is associated with the class $JPS3D$, an object of the class *Geometry* is instantiated with geometry data in the JPS3D constructor. In three.js, the $bufferGeometry$ class represents mesh, line or point geometry efficiently or using a less efficient but easier to use alternative *Geometry* class to create 3D elements. they are defined with vertex positions, surface indices, normals, colors, UVs and user-defined attributes. In the geometry data converted by sever and returned as JavaScript objects, the coordinates of polygons in spaces can be used as vertex positions of $bufferGeometry$. Therefore, the class *Geometry* provides two public methods $createRooms$ and $createTransitions$ which return objects that can be inserted into the scene.

Since the format of the geometry files can be changed in each release version, the version of the geometry files must be validated in the constructor of the *Geometry* class to ensure that JPSvis Online can parse the data correctly. The $createRooms$ method can construct a single room as BufferGeomety, but there is usually more than one room or transition in a geometry, so when $createRooms$ is called, a $three.Grouop$ is created to make working with groups of objects syntactically clearer. When a room

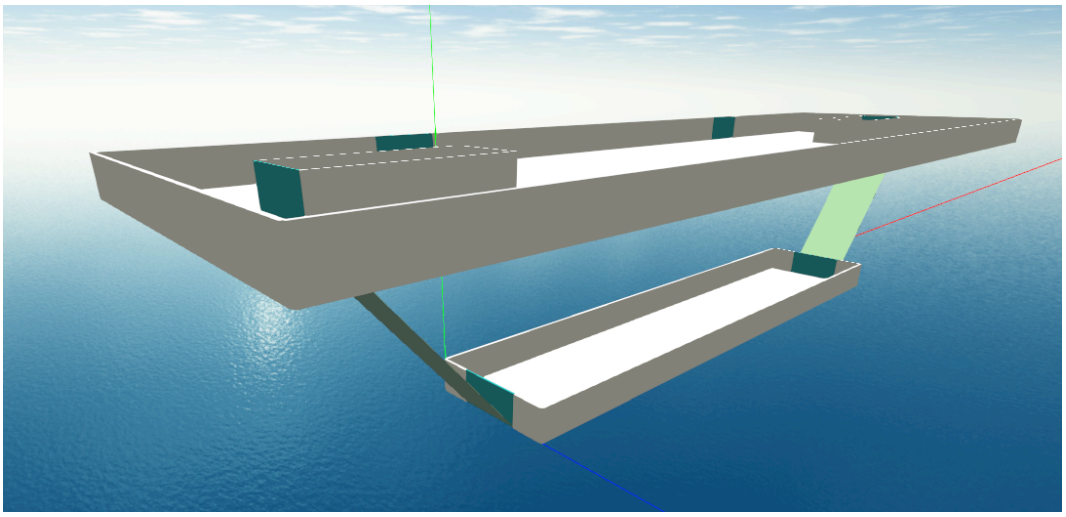


Figure 4.11: A geometry in 3D view

is constructed, each wall in it is treated as $three.BoxBufferGeometry$, which is a rectangular box with a given 'width', 'height' and 'depth'. As discussed in the subsection 3.3.2, the walls in *geometry.xml* are two-dimensional, only the depth can be calculated by the distance of points, so the width and height is given as 1 for all $three.BoxBufferGeometry$. For other elements in the geometry, like platform and transitions, they follow the same path as the room to be created.

Although the data from the geometry file for walls is two-dimensional, the positions of the elements are in three-dimensional space. The Z coordinate of rooms and transitions is stored in the *elevation* property, the elements must be tranlated to the correct position. For stairs there is another problem, the geometry file does not specify the angle of the stairs, so this value must be calculated based on the known parameters, then it rotates itself with the correct angle.

The BufferGeometry only represents the shape of the elements, in three.js there should be a $Meterial$ for a BufferGeometry, which describes the appearance of objects. In geometry there are several elements: room, transition, platform, stair, so for these elements the different materials are designed with $MeshBasicMaterial$, which is a material for drawing geometry in a simple shaded way. Especially for transitions, the opacity is set to fifty percent to show that it is an exit. With BufferGeometry and Materials a $three.Mesh$ can be created, which can be inserted into the scene like the Figure 4.11.

The geometry is defined by the files, but in the *trajectory.txt* there are only the positions of pedestrians, no information about the shape or appearance of pedestrians, so JPSvis Online loads a 3D model of a pedestrian in glTF format according to *loader* class. There are many formats of the 3D model to choose from, glTF focuses on runtime asset delivery, i.e. it can be easily edited in the 3D model editor and in three.js. Its transfers are compact and load quickly. It offers a wide range of functions in the model, including meshes, materials, textures, skins, bones, morphing targets, animations, lights and cameras.

(a) Static pose (b) Walking action

Figure 4.12: The pedestrian 3D model

With the imported model in the Figure 4.12, the model contains a walking animation (see Figure 4.12b, which is useful for simulating the movement within the evacuation process. The animation is not played automatically in the Figure 4.12, but should be controlled by the AnimationMixer, which can control several animations in the model at the same time by mixing and merging them. The AnimationMixer class has very few properties and methods, so it was previously controlled by the AnimationAction class. By configuring an AnimationAction it decides when an animation clip on one of the mixers is played, paused or stopped, how often the clip has to be repeated, and if it has to be hidden or scaled in time. As shown in the Listing 4.8, there is a mixer for each pedestrian to control his animation, and saving the mixers in *this.mixers* as a list. When users click the button to play or pause the animation, an AnimationAction object must be created using the *clipAction*() method, then it can play the animation in the scene. Besides loading the pedestrian model with

glTFloader, the *CubeTextureLoader* is also used to load images of the sky and shows in the scene.

```
setMixer(object: three.Object3D, clip: three.AnimationClip){
    //...
    this.walkingClip = clip;

  const mixer = new three.AnimationMixer( object );
  this.mixers.push(mixer);
}

playAnimation(){
  // Play walking animation
  for(let i=0; i<this.mixers.length; i++){
    const action = this.mixers[i]
        .clipAction(this.walkingClip);
    action.play();
  }
  //...
}
```

Listing 4.8: The animation settings

After loading the 3D model of the pedestrian and the environment for its walking animation, the *updatePedLocation* method is used to update the pedestrian locations. In each frame, the *updatePedLocaion* method updates the position and rotation values from the trajectory data according to the pedestrian index. With continuous updating, the models will reflect the trajectory of the pedestrians walking in the simulation, so in the end the *updatePedLocation* method should be called in the *animate*() method.

The function of the *animate*() method is to create a loop that causes the renderer to draw the scene for each time the screen is updated. The screen refreshes at an unstable frequency, which is influenced by the performance of the devices and contents. So if *animate*() is called at a fixed frequency that does not match the frequency of the screen, the images will collapse. The *requestAnimationFrame* method is a function provided by Web-APIs. It requires the browser to call the specified callback function (here the callback function *animate*()) to update the animation before the next redraw. With this method, a callback function must be passed as argument, which is executed before the next browser redraw. The Figure 4.13 shows the visualization of trajectory files.

```
updatePedLocation(){
  //...
  for(let i=0; i<this.pedestrians.length; i++){
    const id = parseInt(this.pedestrians[i].name);
  const frame = Math.floor(this.frame/8);
  if(frame < this.trajectory.pedestrians[id-1].length){
    const location = this.trajectory.pedestrians[id-1][frame];

    this.pedestrians[i].rotation.y = (location.angle + 90)
      * Math.PI / 180 ;
    this.pedestrians[i].position.x = location.coordinate.x;
    this.pedestrians[i].position.y = location.coordinate.z;
    this.pedestrians[i].position.z = location.coordinate.y;
  }
  }

  this.frame += 1;
}

animate () {
  requestAnimationFrame(this.animate);
    //...
  this.updatePedLocation();
}
```

Listing 4.9: The function for updating pedestrian's location

However, it should be noted that the frame in the screen refresh is different from the frame in the trajectory data. In the simulation the frame is set to eight, i.e. eight snapshots of the position of pedestrians per second, but the browser always tries to update the screens 60 times per second. Therefore *updatePedLocation*() should not take one line in the trajectory file as one frame, but one line across all eight frames, which is as close as possible to the refresh rate of the screen, and the movement of the pedestrian is done at normal speed.

Compared to the UI elements, the pedestrian model consumes a lot of computing power. As the number of pedestrians in the scene increases, the fluidity of the scene is affected. Post-processing is a widely used method to avoid this situation. First, the scene is rendered to a rendering target, which is a buffer in the video card's memory. Then one or more post-processing channels apply filters and effects to the image buffer before the scene is finally rendered to the screen.

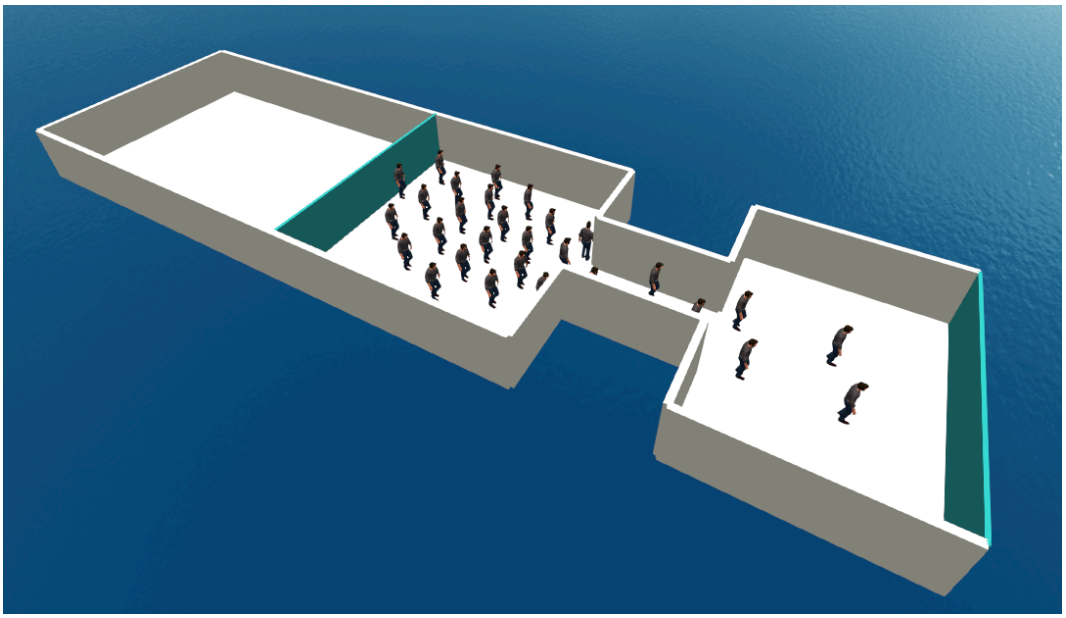


Figure 4.13: The trajectory of pedestrians in 3D view

JPSvis Online implements a complete post-processing solution in the $Effect$ class (see Figure 4.10. The first step in the process is to import all necessary composers and passes from the directory. For JPSvis Online the $EffectComposer$ and $SMAAPass$ are used. $EffectComposer$ manages the chain of postprocessing processes and creates the final visual effects. The post-processing processes are executed in the order in which they are added or inserted and the last process produces a result that is automatically displayed on the screen. $SMAAPass$ adds an anti-aliasing effect to the scene.

Figure 4.14: The menu for 3D view

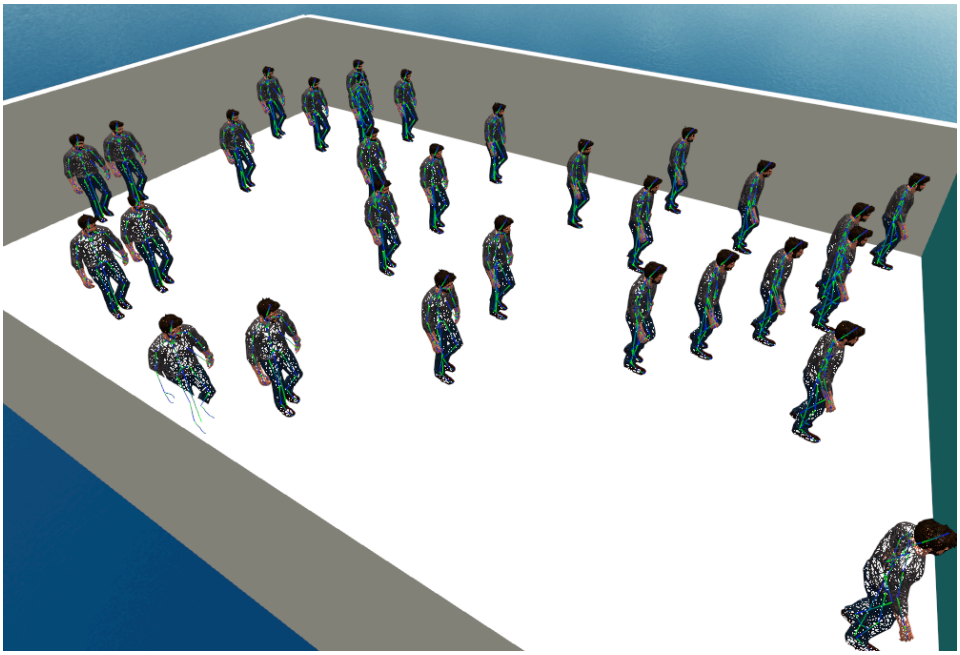


Figure 4.15: The skeleton view of pedestrian model

The last modules in the JPS3D class are methods that react to user interaction. As can be seen in the Figure 4.14, the first folder should control the display of pedestrians with checkbox. By default, pedestrians are not displayed by opening the 3D view, users can view the geometry without being distracted by pedestrian models. The wireframe and skeleton options show the structure of the pedestrian model as in Figure 4.15, so that users can better observe the actions of pedestrians. The second folder contains the game options. When users click buttons, the *playAnimation*, *pauseAnimation*, and *resetPedLocation* methods are called to play, pause, and reset pedestrian position updates.

```
switchTo2D () {
  const canvas = document.getElementsByTagName('canvas');
  for(let i = canvas.length - 1; i>=0; i--){
    canvas[i].parentNode.removeChild(canvas[i]);
  }

  (async () => {
    const initResources = await init();
    const jps2d = new JPS2D(initResources.geometryRootEl, initResources)
  }
  )();
}
```

Listing 4.10: Switch to 2D view

The last button on the menu is "Switch to 2D". The Listing 4.10 shows the source code for it. First, the *canavas* tag is removed from the HTML tree, i.e. all content, including geometry, lights and pedestrians, even the scene of the 3D view, disappears. Then the *new* keyword is used to create an instance of $JPS2D$, the 2D view appears and replaces the content of three.js.

4.2.4 2D visualization

This section describes how to display the 2D view, most programming logics in it are very similar to the 3D view, but with the PixiJS library. Therefore the focus here is on explaining how it differs from the 3D view.

The initialization of the 2D-view is done in the function $switchTo2D$, which is similar to the initialization of the 3D-view. The preparation consists of waiting for the *init*() response with the *async* function, then adding the *canavas* tag, and the geometry and path data is fetched in the *init*() function and stored in the *initResources* function, which is a parameter for the constructor of the $JPS2D$ class. After instantiating the $JPS2D$ class, PixiJS starts to display the content under the *canavas* tag. The class diagram of $JPS2D$ shows in the Figure 4.16

Similar to rendering the 3D view, the first step in rendering the 2D view is to create a PixiJS application. It will automatically create a $<$ *canvas* $>$ HTML tag and figure out how to display the data in that tag. All contents are displayed by PixiJS *Container*, the initial container called *stage*, which is used as root container that wraps all displays with PixiJS. To display the PixiJS content in the browser, add the canvas that PixiJS has automatically created to the HTML document. $PIXI.Application$ specifies whether the browser should use Canvas or WebGL to render the content, depending on which technology the browser supports.

With the *stage* container it is possible to add graphical elements to show geometry, the geometry is drawn with the $Geometry2D$ class (see Figure 4.16). All shapes in PixiJS are initialized by first creating an instance of the $Pixi.Graphics$ class, there is the method lots to draw different shapes. For example the $drawCircle$ for circle, $drawRect$ for rectangles. As can be seen in the Listing 3.1, rooms or transitions are polygons, which are actually a set of lines, so the $addRoom$ and $addTransition$ can call the $createLine$ method to compose room and transition with different colors.

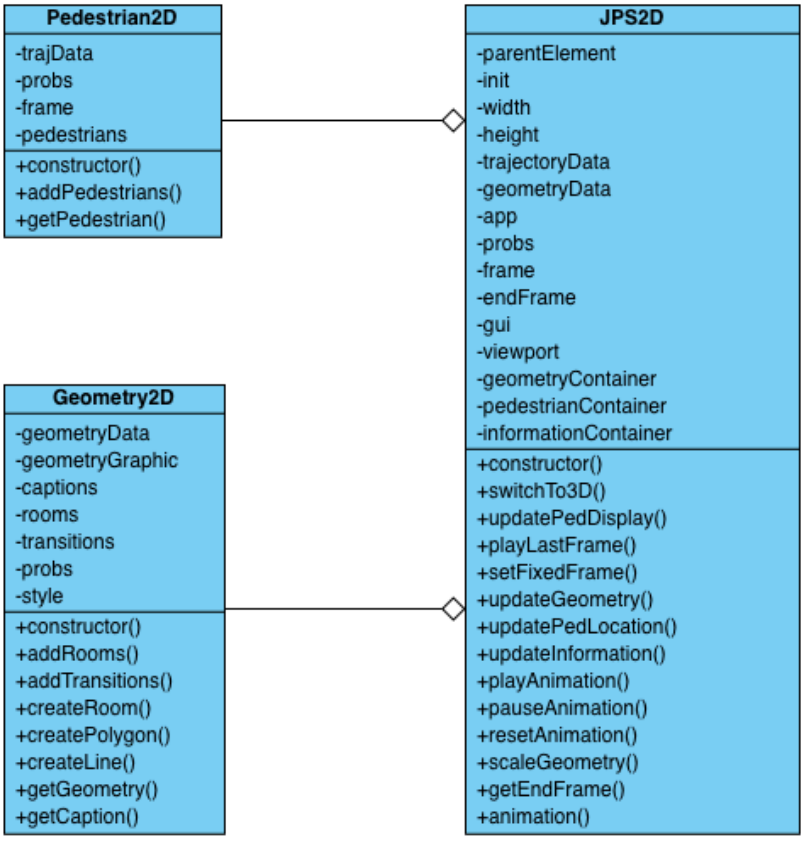


Figure 4.16: Class diagram for 2D view

```
createLine(element, color){
  const point1x = parseFloat(element.vertex[0].px)
    * this.probs.scale + this.probs.offsetX ;
  const point1y = parseFloat(element.vertex[0].py)
    * this.probs.scale + this.probs.offsetY ;
  const point2x = parseFloat(element.vertex[1].px)
    * this.probs.scale + this.probs.offsetX ;
  const point2y = parseFloat(element.vertex[1].py)
    * this.probs.scale + this.probs.offsetY ;

  this.geometryGrapihc.beginFill();
  this.geometryGrapihc.lineStyle(1, color, 1);
  this.geometryGrapihc.moveTo(point1x , point1y);
  this.geometryGrapihc.lineTo(point2x, point2y);
  this.geometryGrapihc.endFill();
}
```

Listing 4.11: Draw a line in PixiJS

However, the *Pixi.Graphics* class draws elements as the unit, and the usual screen resolution is 1920x1080. If the original geometry data is entered directly as the length of the lines, the geometry will be very small on the screen. Therefore it is necessary to scale the size of the geometry, by default the scaling rate is 10, but in the meantime the geometry must be offset to ensure that it appears as far as possible in the center of the screen, since the geometry often starts around origin, which is located in the upper left corner of the screen. To make scaling and offsetting easier to adjust, all the geometry is drawn in a single *Pixi.Graphics* object (see *this.geometryGrapihc*) in the Listing 4.11.

To represent pedestrians, *Pedestrian2D* also uses the *Pixi.Graphics* class to draw circles representing pedestrians. Unlike the *Geometry2D* class, the position of a pedestrian is independent of others, so if all pedestrians are drawn by a single *Pixi.Graphics* object, it is impossible to update their positions independently, so for each pedestrian a new *Pixi.Graphics* is created to draw a circle and all pedestrians are added to a specific container in *this.pedestrians*. After adding geometry and pedestrians in the constructor of class *JPS2D*, the 2D view will be shown as theFigure 4.17 No.1.

PixiJS is a rendering library that allows developers to create rich, interactive graphics. However, it lacks a control component to adjust the display, e.g. zoom in and out. Therefore the pixi-viewport library is introduced. The pixi-viewport is a highly

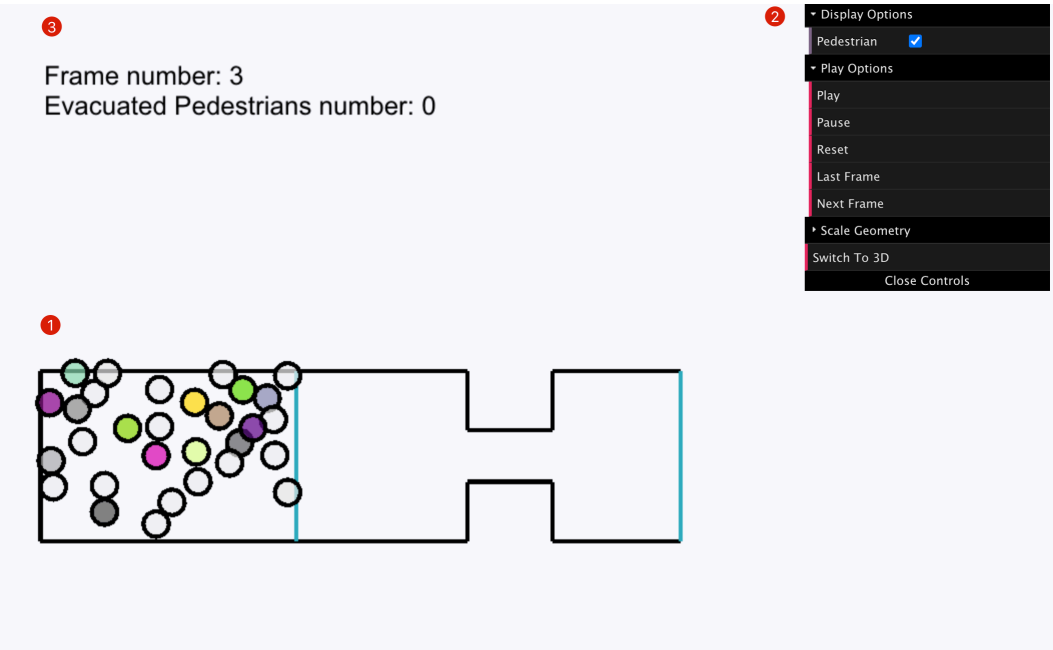


Figure 4.17: The 2D view

configurable 2D camera, which was developed for working with pixi.js. The functions include dragging, pinch-to-zoom, mouse-wheel zooming, etc. To work with pixi-viewport, the Pixi's container must be added as a child to the *Viewport* class, then these containers and their graphics are movable and configurable.

To show the trajectory of pedestrians, the mechanism is similar for the 3D-viewport. The movement of pedestrians is defined by the coordinates in the trajectory file, the *updatePedLocation*() method reads the exact X and Y coordinates, and the *animate*() method updates the position of pedestrians in each frame. As in the 3D view, the *requestAnimationFrame*() method is used to update the animation before the next redraw, and every eight frames the *updatePedLocation* method updates the position of pedestrians. In the trajectory file there is a number indicating the color of a pedestrian, in 2D view it is used to fill in the color of circles (see Figure 4.17 no.1).

In the 2D view there is also a menu to interact with the content (see no.2 the Figure 4.17). As in the 3D view, there are *Play*, *Pause* and *Reset* buttons to start, pause and reset the animation. To view the animation in a fixed frame, there are *LastFrame* and *NextFrame* to control the frame number. As in the Figure 4.17 No.3, it counts the frame number and the number of pedestrians evacuated, which means that its position is not within the geometry and will not change.

Option	Value
N-t Diagram	N_t
Density - Time	Density_Time
Velocity - Time	Velocity_Time
Density - Velocity	Density_Velocity
Density - Specific Flow	Density_Flow
Density Spatiotemporal Profile	Profiles_Density
Velocity Spatiotemporal Profile	Profiles_Velocity

Table 4.1: Affordable APIs

The last module in the 2D view is $Switchto3D$, it removes the content placed under the *canvas* tag and creates an instance of $JPS3D$, the 3D view appears at the position where the 2D view is located.

4.2.5 Analysis module

The analysis module in the JPSvis Online based on the scripts of the JPSreport, but has been edited to fit the architecture of the client-server model. As discussed in the subsection 3.2.2, the server's task is to provide APIs to answer the request, the call and return of the analysis script is also done via APIs.

As shown in the Figure 4.9a, the user could select a diagram type from the menu, with each option in the menu representing a value that can call an HTTP request. The Table 4.1 shows all values representing options.

Before the scripts for generating diagrams are executed, the user must upload the correct output files using the upload component on the view page (see no.1 in the Figure 4.8). For the diagram shown in the Table 4.1, the following Table 4.2 contain all types of output files from JPSreport that can be used for plotting diagrams. Methods C and D calculate data using the classic method and the Voronoi method, respectively, but they can use the same APIs for plotting diagrams because their output files are stored in the same format. To get and read these uploaded files, they are also renamed by the function $post\_file$.

When the user clicks on an option, the browser requests a response. For example. If the "Density Profile" is selected, the request is "http://localhost:8080/IFD_traj". As can be seen from the Listing 4.1, there are already many handlers that deal with

Diagram	File Name	New Name	Method
N-t Diagram	Flow_NT_traj	N_t.dat	A
Density - Time	rho_v_*_traj_	rho_v.dat	C/D
Velocity - Time	rho_v_*_traj_	rho_v.dat	C/D
Density - Velocity	rho_v_*_traj_	rho_v.dat	C/D
Density - Specific Flow	rho_v_*_traj_	rho_v.dat	C/D
Density Profile	IFD_traj	IFD.dat	D
Velocity Profile	IFD_traj	IFD.dat	D

Table 4.2: Outputs file for generating diagram

these requests. This request will call the function *get_profile_density* (see in the Listing 4.12).

The function first fetches the necessary information, for the display of the density profile it needs geometry, trajectory and IFD files. Then these files are sent as parameters to the function responsible for plotting. These scripts are stored in the *analysis* folder on the server. After executing the function, the diagram is saved in the *server* folder with a formatted name, for the density profile it is called "profile_density.png".

```
async def get_profile_density(request):
    geofile = 'geometry.xml'
    trafile = 'trajectory.txt'
    IFDfile = 'IFD.dat'

    if not os.path.exists("profile_density.png"):
        plot_profiles.plot_profiles(geofile, trafile, IFDfile)

    with open("profile_density.png", "rb") as img_f:
        return web.Response(
            text=base64.b64encode(img_f.read()).decode('utf-8'))
```

Listing 4.12: The function for plotting density profile

```
<Modal title={this.state.url} visible={this.state.visible}
    onOk={this.handleOk}
    onCancel={this.handleCancel} width={700}>
    <img src ={""
        + this.state.imgData} width={500}/>
</Modal>
```

Listing 4.13: The Modal window for showing plots

The next step is to return the image to the browser to display it in the window shown in the Figure 4.9b. There are some approaches to transfer images from the server to the browser, since JPSvis Online only displays one image at a time. The more direct way to transfer images is to convert the image into base64 code and return the text code with HTTP response. Base64 is a representation method for displaying binary data based on 64 printable characters, with the function $base64.b64encode$ an image can be encoded as text data so that it can be transferred into an HTTP response. On the browser view page the browser can directly decode base64 text with *img* tag, so that as shown in the Listing 4.13 the diagram data is stored in $this.state.imgData$ wrapped in the *img* tag, and the modal window displays it as Figure 4.18 with 500px width.

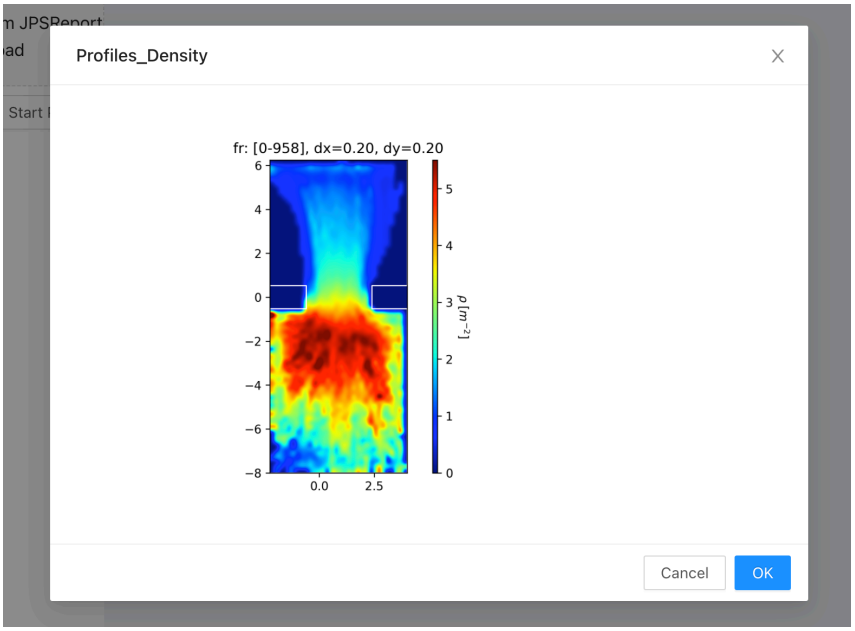


Figure 4.18: The density profile

Chapter 5

Tests

This chapter describes the tests for JPSvis Online. The tests can be divided into the product test and the acceptance test. The product tests focus on the complete functionality of the product, checking whether the product meets the requirements listed in the chapter 3. The goal of the acceptance tests is to determine whether the product actually meets the goals set by the project.

5.1 Test cases

The product test is performed with test cases, whereby the test cases are divided into two groups: functional execution and error handling. The functional execution test cases contain the planned user input and the expected reaction of the application. The error handling test cases describe how JPSvis Online handles the error and returns the message to the user.

The test cases are organized and designed according to scenarios, in each scenario there is a task for functional execution and error handling. The test cases use a white box test system, i.e. they are designed with the scenarios of JPSvis Online in mind. The goal of these tests is to ensure the functionality of all features implemented in this thesis. Errors handling are handled appropriately to avoid unexpected behaviours. If the behavior of JPSvis Online matches the expectations, the test cases lead to success.

5.2 Server hosting

Test scenario: Hosting a sever on a specific port, and starting the JPSvis Online application in the *index.html*

User input: Running the *server.py* script and input $http://localhost:8080/$.

Expected behavior: Showing a message in the terminal when server is hosted successfully, and the JPSvis Online is showing in a browser.

Result: The terminal shows the message as the Figure 5.1, the website is opened successfully on the port 8080 as except, it can be visited until the server is shut down.

Error handling: The browser shows "500 Server Error" when JPSvis Online isn't loaded successfully.

Figure 5.1: The message in the terminal

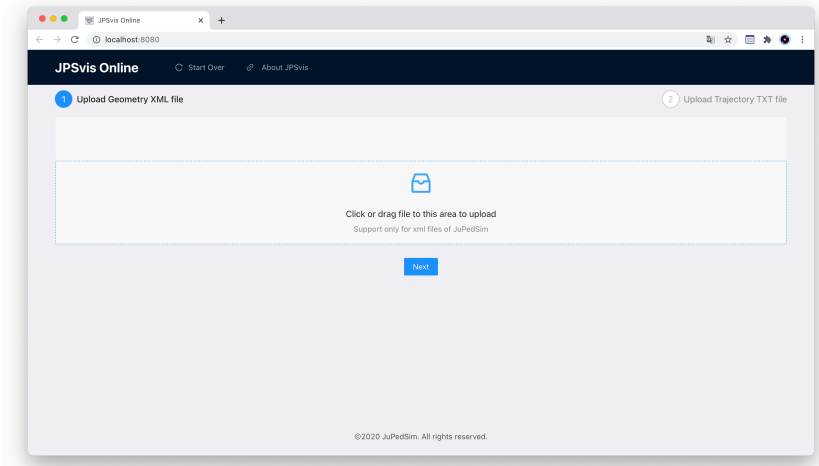


Figure 5.2: Successful website opening

5.3 Uploading files

Test scenario: Uploading a geometry (XML) and a trajectory file (TXT) to the sever.

User input: Clicking the uploading area in the uploading page, selecting a file (geometry in the first step and the second step for trajectory) to upload.

Expected behavior: The uploaded files should be saved into the *server* folder in the server, and they should be renamed as *geometry.xml* and *trajectory.txt*. On the uploading page there will be a message to show that the process of uploading is successful.

Result: The process of uploading is successful. The message show in the Nr.1 in the Figure 5.3.

Error handling: If the format of files is invalid, declining to upload and show error message as the Figure 5.4.

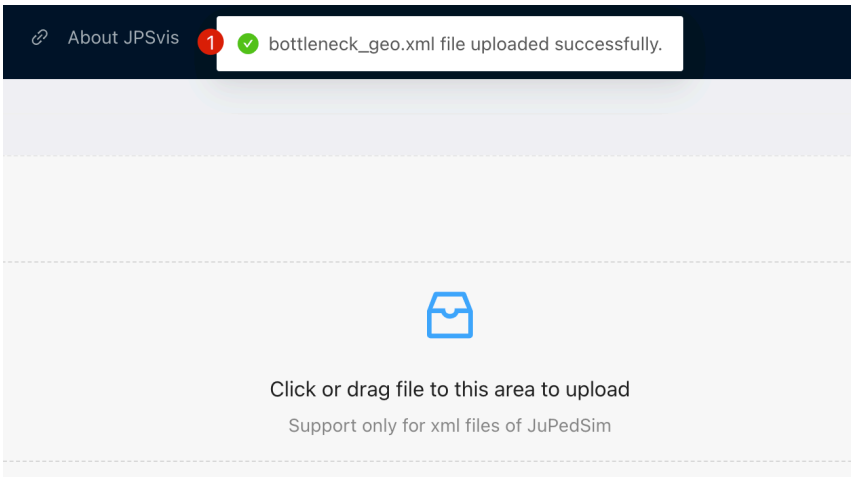


Figure 5.3: The message after successful uploading

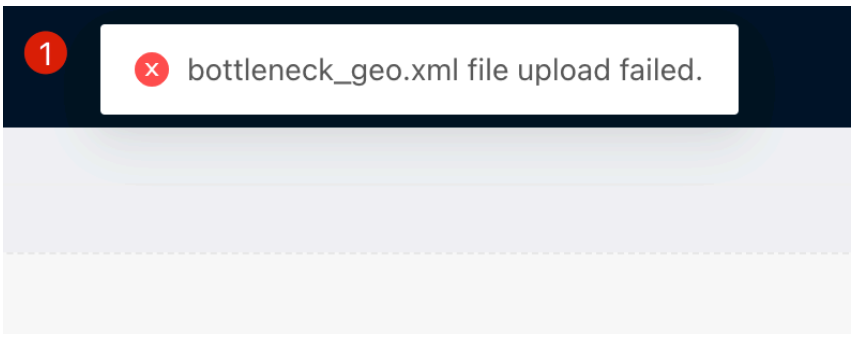


Figure 5.4: The message after failed uploading

5.4 Starting 3D view

Tested functionality: Starting presentation of geometry in 3D view

User input: When the view page is opened, the user doesn't nee to do any input, it should append automatically. If in the 2D view, the 3D view should be opened after clicking the *Switchto3D* button.

Expected behavior: The JPSvis Online can show the geometry file with 3D element completely.

Result: Viewing the geometry in 3D is successful as show in the Figure 5.5.

Error handling: If the contents of geometry is invalid and incomplete, the 3D view should present as much as possible.

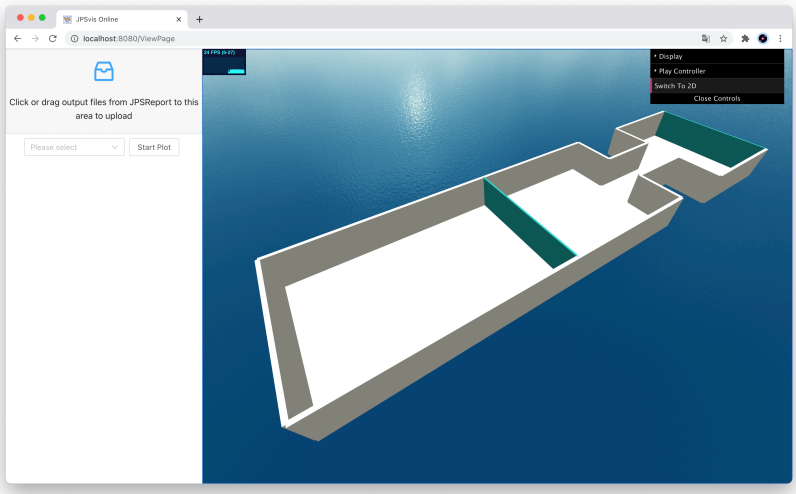


Figure 5.5: The 3D View

5.5 Interacting with mouse in the 3D view

Tested functionality: The controlling of mouse should work as designed.

User input: Dressing the left button and moving mouse to adjust the location of camera of view; rotating the wheel to zoom in and zoom out, changing the size of geometry in the view; dressing the right button and moving mouse to change the angle of the camera.

Expected behavior: The 3D view changes itself smoothly according to operations of the mouse movement.

Result: Success. The mouse interacts with 3D view without decreasing the FPS of the view.

5.6 Interacting with menu in the 3D view

Tested functionality: The options in the menu should work as designed.

User input: Changing the display of pedestrians through selecting (or not) the options under the *Display* folder; Controlling the animation of trajectory by buttons under *PlayController*; Clicking the *SwitchTo2D* button to start 2D view.

Expected behavior: The 3D view changes itself smoothly according to options in the menu.

Result: Success. The menu interacts with 3D view without decreasing the FPS of the view.

5.7 Starting 2D view

Tested functionality: Starting presentation of geometry in 2D view.

User input: The user clicks the *SwitchTo2D* button in the menu of 3D view.

Expected behavior: The 3D view disappears and the 2D view presents itself.

Result: Viewing the geometry in 3D is successful as show in the Figure 5.6.

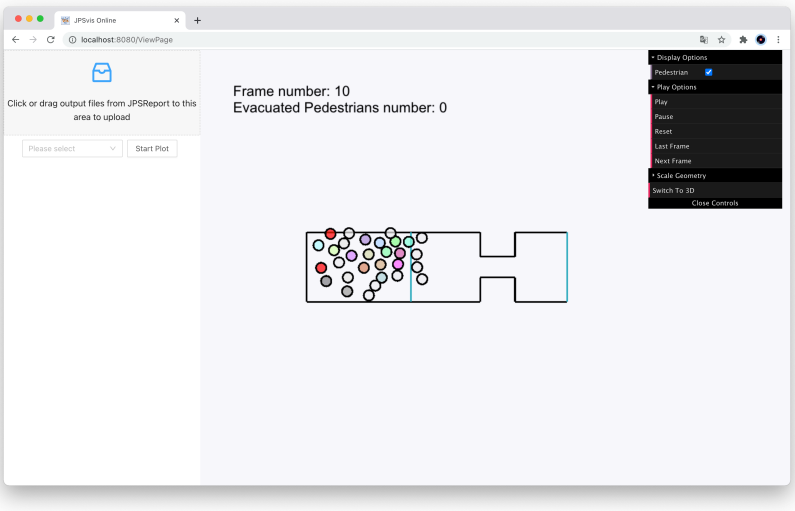


Figure 5.6: The 2D View

5.8 Interacting with mouse in the 2D view

Tested functionality: The controlling of mouse and buttons should work as designed

User input: Dressing the left button to move the geometry; rotating the wheel to zoom in and zoom out.

Expected behavior: The 3D view changes itself smoothly according to operations of the mouse movement.

Result: Success. The mouse interacts with 2D view successfully

5.9 Interacting with menu in the 2D view

Tested functionality: The options in the menu should work as designed.

User input: Controlling the animation of trajectory by buttons under *PlayController*; Scaling the geometry with *scale* toolbar; Clicking the *SwitchTo3D* button to start 3D view.

Expected behavior: The 2D view changes itself smoothly according to options in the menu.

Result: Success. The menu interacts with 2D view successfully

5.10 Plotting diagram

Tested functionality: The analyzsis scripts plot the diagram after receiving the message from the selecting menu.

User input: The user should choose a type of diagram in the selecting menu, then clicking the *StartPlot* button.

Expected behavior: A modal window will show up in the viewing page to show the diagram.

Result: Success. The Figure 5.7 shows the modal window and the diagram.

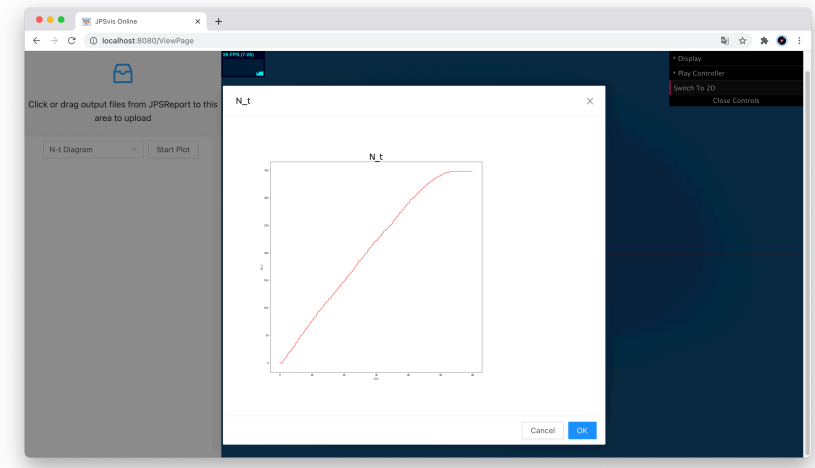


Figure 5.7: Plotting diagram

Chapter 6

Summary and Outlook

The purpose of this thesis is to develop JPSvis Online, a web-based tool used within JuPedSim for visualization and analysis of pedestrian traffic simulation results. To achieve this target, the requirements are well defined before implementation with five objectives. This chapter describes the main features and constraints as well as possible orientations for future developments.

6.1 Summary of JPSvis Online

The development of JPSvis Online follows the sever-client model, in which the structure of the server forms the basis for other components, e.g. UI and analysis. The server for JPSvis Online is based on the package aiohttp in python. It helps developers to build an asynchronous HTTP server. Since aiohttp and its analysis scripts are both written in python, it has a better compatibility than other HTTP server frameworks written in Java or JavaScript. With the help of the asynchronous structure, the server of the JPSvis can handle the operation challenge caused by high concurrency.

The main task of the JPSvis server is to respond to various HTTP requests using many APIs. Basically, the APIs can be grouped into three categories: handlers for switching websites, handlers for returning geometry and trajectory data in JSON format to the browser, and handlers for generating diagrams.

After the server is implemented, the UI is created with the React framework in JPSvis Online. The UI of JPSvis Online is designed with the upload page and the view page. The upload page allows users to submit the geometry and trajectory files

to the server so that JPSvis can parse and visualize them. The view page consists of three parts: the component for uploading output files from JPSreport, the drop-down menu for selecting the type of diagram that the user would like to generate, and the opinion designed for 2D and 3D view. Apart from the above components, a modal window appears when a diagram is generated.

As the first step of visualization, parsing for geometry and trajectory file is implemented. Once the geometry file (XML format) and the trajectory file (txt format) have been uploaded, they are converted and stored in dictionaries using Python scripts. When the browser calls these files, they are converted as text in JSON format and returned with an HTTP response.

In the 2D visualization, the PixiJS framework is used to represent the geometry of the building. Rooms, stairs and transitions are all drawn directly after the geometry file. At the same time, the pedestrians are displayed as a circle with color, which indicates the velocity. In case of the 3D view, the three.JS is is used to visualize the geometry and trajectory file. The position, width and height of the walls are calculated from the original data. The 3D model of the pedestrians is loaded from a glTF file.

Whether in 2D or 3D view, the animation of pedestrians is implemented with a similar logic. JPSvis Online updates the position of pedestrians in each frame and renders the scene in real-time. Therefore, the process of simulation in JPSvis Online is displayed in view of the users. When interacting with the geometry and the pedestrians, the users can control the animation of the pedestrians and the view of the geometry with mouse and touchscreen. There is a menu in the 2D and 3D view that allows users to select the content to be displayed in the view.

For the purpose of analysis, the user can create a variety of diagrams in JPSvis. For instance, the Density frame or Velocity frame diagram shows the changes in density or velocity during the simulation, the Density-Velocity fundamental diagram and the Density-Specific Flow diagram indicate the relationship between density, velocity and the specific flow. The profiles of density and velocity intuitively illustrate the variability over a given time period.

Finally, several tests are performed for JPSvis Online to ensure that the functional modules in the software work properly. These tests not only reveal the necessary error handling, but also check whether the software has achieved what it was designed for.

6.2 Outlook

With the effort in this thesis, the set objectives are achieved by developing the new software JPSvis Online within JuPedSim. However, there are still some limitations in the current JPSvis Online and some features may be added in the future development.

Currently, JPSvis Online offers only a preliminary capability to display 3D geometry with limited accuracy and type of geometry. In further work, this can be improved with the use of high-resolution materials. For example, the interface of walls and transitions can be decorated with real materials and the design of stairs can be optimized closer to the real building. With these improvements, the reproducibility of the scene and the facilitation can contribute to a better visual representation of the experiment.

For visualization purposes, users have to upload geometry and trajectory files. In fact, the experiment simulation task can also be integrated into JPSvis Online, so that the user only has to upload the geometry file and the ini-files for JPScore and JPSreport. In this way, JPSvis Online can run the simulation directly by calling JPScore on the server or locally. Additionally, the task of trajectory analysis in JPSreport can also be shifted to JPSvis Online, which can reduce the operation time in the browser for the user significantly.

Until now, JPSvis Online can only generate static diagrams in the browser, so users cannot create them interactively. In the coming development, the analysis module can be extended to be an interactive module that allows users to change the area of interest in the diagram making it possible to perform analysis and research in real time.

Despite being designed as a client-server model, the current JPSvis Online runs on the local computer of the development environment. Therefore, users have to start the server on their own computer and visit the browser on the same computer. If JPSvis Online is deployed on a public server, users can visit the server on any device that supports modern browsers and do not need to start the server locally.

Bibliography

[1] Erik Andresen. *Wayfinding and Perception Abilities for Pedestrian Simulations*. Number FZJ-2019-00686. Zivile Sicherheitsforschung, 2018.

[2] Alex Banks and Eve Porcello. *Learning React: functional web development with React and Redux*. " O'Reilly Media, Inc.", 2017.

[3] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

[4] Daniel Büchele and Mustafa K Isik. Webbasierte 3d-postvisualisierung von fußgängersimulationsdaten. 2012.

[5] Winnie Daamen. Modelling pedestrians in transfer stations. In *Five Years Crossroads of Theory and Practice. Proceedings 5th TRAIL Annual Congress 1999Netherlands TRAIL Research School*, number Part 2, 1999.

[6] Jos Dirksen. *Learning Three. js: the JavaScript 3D library for WebGL*. Packt Publishing Ltd, 2013.

[7] Lee Zhi Eng. *Qt5 C++ GUI programming cookbook*. Packt Publishing Ltd, 2016.

[8] S Faulkner, A Eicholz, T Leithead, A Danilo, and S Moon. Html 5.2 specification, 2018.

[9] David Flanagan. Javascript: the definitive guide, 2006.

[10] John J Fruin. Pedestrian planning and design. Technical report, 1971.

[11] David Halliday, Robert Resnick, and Jearl Walker. *Fundamentals of physics*. John Wiley & Sons, 2013.

[12] Charles D Hansen and Chris R Johnson. *Visualization handbook*. Elsevier, 2011.

[13] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90–95, 2007.

[14] Mustafa K Isik. Sumoviz, html5-based visualization of pedestrian simulation data. 2012.

[15] Armel Ulrich Kemloh Wagoum, Mohcine Chraibi, and Gregor Lämmel. Jupedsim: an open framework for simulating and analyzing the dynamics of pedestrians.

[16] Ansgar Kirchner, Hubert Klüpfel, Katsuhiro Nishinari, Andreas Schadschneider, and Michael Schreckenberg. Discretization effects and the influence of walking speed in cellular automata models for pedestrian dynamics. *Journal of Statistical Mechanics: Theory and Experiment*, 2004(10):P10011, 2004.

[17] Wen Syan Li, Kasim Selouk Candan, and Divyakant Agrawal. System and method for intelligent caching and refresh of dynamically generated and static web content, July 8 2003. US Patent 6,591,266.

[18] Jeanine Meyer and Corrigan. *Essential Guide to HTML5*. Springer, 2018.

[19] Tim Meyer-König, Hubert Klüpfel, and Michael Schreckenberg. Assessment and analysis of evacuation processes on passenger ships by microscopic simulation. *Schreckenberg and Sharma [2]*, pages 297–302, 2002.

[20] Jeffrey C Mogul, Roy Fielding, Jim Gettys, and Henrik Frystyk. Rfc2145: Use and interpretation of http version numbers, 1997.

[21] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of json and xml data interchange formats: a case study. *Caine*, 9:157–162, 2009.

[22] Haroon Shakirat Oluwatosin. Client-server model. *IOSR J Comput Eng (IOSR-JCE)*, 16(1):67, 2014.

[23] Vitaly M Predtechenskii and Anatoliĭ Ivanovich Milinskiĭ. *Planning for foot traffic flow in buildings*. National Bureau of Standards, US Department of Commerce, 1978.

[24] Paul Read and Mark-Paul Meyer. *Restoration of motion picture film*. Elsevier, 2000.

[25] Andreas Schadschneider, Wolfram Klingsch, Hubert Klüpfel, Tobias Kretz, Christian Rogsch, and Armin Seyfried. Evacuation dynamics: Empirical results, modeling and applications. *arXiv preprint arXiv:0802.1620*, 2008.

[26] Armin Seyfried, Bernhard Steffen, and Thomas Lippert. Basics of modelling the pedestrian flow. *Physica A: Statistical Mechanics and its Applications*, 368(1):232–238, 2006.

[27] TN Sharma, Priyanka Bhardwaj, and Manish Bhardwaj. Differences between html and html 5. *International Journal Of Computational Engineering Research*, 2(5):1430–1437, 2012.

[28] Julia Valder. Developing a multiuser platform for virtual experiments in pedestrian dynamics. Technical report, Jülich Supercomputing Center, 2016.

[29] Rex Van der Spuy. *Learn Pixi. js*. Apress, 2015.

[30] Guido Van Rossum and Fred L Drake. *The python language reference manual*. Network Theory Ltd., 2011.

[31] Lauren Wood, Arnaud Le Hors, Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Ian Jacobs, Gavin Nicol, Jonathan Robie, Robert Sutor, et al. Document object model (dom) level 1 specification. *W3C recommendation*, 1, 1998.

[32] Jun Zhang. *Pedestrian fundamental diagrams: Comparative analysis of experiments in different geometries*, volume 14. Forschungszentrum Jülich, 2012.