



EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

DEPT. OF PROGRAMMING LANGUAGES AND COMPILERS

SmartNav: Indoor Navigation Mobile Application

Supervisor:

Morse Gregory

Phd Student and Teaching assistant

Author:

Abdelaziz Dhaouadi

Computer Science BSc

Budapest, 2024

Thesis Registration Form

Student's Data:

Student's Name: Dhaouadi Abdelaziz

Student's Neptun code: B0PQRW

Course Data:

Student's Major: Computer Science BSc

I have an internal supervisor

Internal Supervisor's Name: *Morse Gregory Reynolds*

Supervisor's Home Institution:

Department of Programming Languages and Compilers

Address of Supervisor's Home Institution:

1117, Budapest, Pázmány Péter sétány 1/C.

Supervisor's Position and Degree:

Msc in Computer Science, PhD Student and Teaching Assistant

Thesis Title: Indoor Navigation Mobile Application

Topic of the Thesis:

(Upon consulting with your supervisor, give a 150-300-word-long synopsis of your planned thesis.)

Indoor Navigation System for Elte's faculty of informatics:

The main objective of this project is to create SmartNav which is a 3D mobile application that provides a visualization of the faculty maps and create an indoor navigation system. Users will be able to reach their designated facility such as classrooms, libraries, restrooms and cafeteria in an efficient way.

Users will mention the entrance from which they got into the faculty, this determines the floor they are actually on. Moreover, floors will be represented separately which means changing floors either on foot or using the elevator will take you to the next one.

A floor will be presented as 3D labeled cuboids for the rooms and facilities and stairs as diagonal ramps.

The maps of the faculty as a data source, delivered as pictures to the app, will be treated in a way that flood fill algorithm is used to create polygons which are used within the computational geometry for the classrooms and the facilities of the faculty.

However, elevators, stairs and every door of the classroom will be manually identified.

The obtained polygons are used to implement a shortest path algorithm that provides the required shortest distance.

SmartNav will provide the users with three different modes of use:

Normal navigation mode, emergency mode to help find the nearest emergency exit in case of unexpected situations and for students with health issues who need a way to get around the faculty with less effort a special demand mode is designed.

Budapest, 2023. 12. 01.

Acknowledgements

First and foremost, I'd like to express my deep gratitude to my parents and sister for their support and encouragement along the way. Their sacrifices and help have helped shape the person I am today.

I am immensely grateful to my supervisor, Professor Morse Gregory, who believed in my abilities and supported me relentlessly, pushing me to go beyond my limits. His valuable discussions and advice were crucial to the completion of this project.

I would also like to thank the Geoinformatics Department at the University of Elte, and in particular Professor Gede Matias, for their help in providing me with the data I needed to complete this project. Their support was essential for the realization of this project.

Dédicaces

Avant tout, je voudrais exprimer ma profonde gratitude à mes parents et à ma sœur pour leur soutien et leurs encouragements tout au long de ce parcours. Leurs sacrifices et leur aide ont contribué à forger la personne que je suis aujourd'hui.

Je suis immensément reconnaissant à mon encadrant, le professeur Morse Gregory, qui a cru en mes capacités et m'a soutenu sans relâche, en me poussant à dépasser mes limites. Ses discussions et conseils précieux ont été cruciaux pour l'achèvement de ce projet.

Je tiens également à remercier le département de Géo-informatique de l'Université d'Elte, et plus particulièrement le professeur Gede Matias, pour l'aide qu'ils m'ont apportée en me fournissant les données nécessaires à la réalisation de ce projet. Leur soutien a été essentiel pour la réalisation de ce projet.

Contents

Acknowledgements	2
1 Introduction	3
2 User documentation	4
2.1 System requirements	4
2.2 Installation guide	5
2.2.1 Software requirements and Dependencies	5
2.3 App Launch	8
2.3.1 Loading screen	9
2.3.2 Mode-selection screen	10
2.3.3 Destination-selection screen	11
2.3.4 Map screen	19
3 Developer documentation	26
3.1 Design	26
3.1.1 Introduction	26
3.1.2 System Architecture	27
3.1.3 Application Layout	28
3.1.4 Data Sources	31
3.1.5 3D Modeling	35
3.1.6 2D Modeling	38
3.1.7 Shortcomings	39
3.2 Implementation	41
3.2.1 Introduction	41
3.2.2 Data Processing	41
3.2.3 2D Implementation	51
3.2.4 3D Implementation	53

3.2.5	Rendering	56
3.2.6	Events Handling	59
3.2.7	Shortcomings	60
3.3	Testing	61
3.3.1	Performance Testing	61
3.3.2	Manual Testing	66
4	Conclusion	71
	Bibliography	72
	List of Figures	74
	List of Tables	76
	List of Algorithms	77
	List of Codes	78

Chapter 1

Introduction

The difficult design of the faculty building is a typical complaint among former students of Eötvös Loránd University. Its several levels and intricate hallways make it a struggle to get to the targeted spots. Because of this complicated design, people usually arrive late to class or, worse, miss the start of their exam. The complex layout not only makes studying more stressful, but it also emphasizes how important it is to have better directional signs and directional assistance on university property.

Here is where the idea of creating a mobile app originated from, due to the fact that students would rather have a map aide on their phones than use antiquated wall maps.

SmartNav is a 3D mobile application developed to facilitate indoor navigation for students within the Faculty of Informatics at Eötvös Loránd University (ELTE). It addresses the challenge faced by newcomers in navigating the South Building, assisting in locating classrooms, restrooms, cafeteria and libraries by calculating the shortest path to the desired destination. The app is designed with simplicity in mind, ensuring that users can find their destinations with just a few clicks. It offers different modes, including options for students with disabilities, making it accessible to everyone. This straightforward approach makes navigation easy and straightforward for all users. The app not only addresses the navigation challenges but also prioritizes user safety by including an Emergency mode. This feature is designed to quickly guide users to the nearest exit in case of an emergency.

Chapter 2

User documentation

The design of the app is centered on user-friendliness, with the tagline "Simplicity is user-friendly." It effectively solves the complicated design of the faculty building through a clean and straightforward interface.

The app's maps may not be entirely dependable because they may be out-of-date or have problems with certain parts, such as : the main stairs of the building are missing, missing elevators. The application will, however, offer detours to compensate for the real-world shortest way. This makes it unreliable for determining the shortest path, as it is in real life.

2.1 System requirements

The target users of the application are android owners since it is only supported by Android with a minimum software requirement for the application to run, as it is supposed to, of Android 5.0 Lollipop.

The application is made to be both inclusive and flexible; it works perfectly with both armeabi-v7a [1], arm64-v8a [2] ABI. Yet a minimum of 1 GHz 32-bit (x86) or 64-bit arm (x64) CPU, is advised, alongside with a 2 GB RAM for a smooth experience of the app.

For the needed memory space, a minimum of 90 MB will be needed for the APK file. However, after the installation takes place, the app will require 240 MB of memory.

2.2 Installation guide

If your device meets the minimum requirements of the application, and you would like to give it a try, the creation of the Android Package Kit (APK) file can be done the following way:

2.2.1 Software requirements and Dependencies

Software requirements

Before you begin, ensure that your development environment meets the following requirements:

- Python 3.x [3]
- Python package manager (pip) [4]
- Ubuntu [5]/Debian-based systems [6]

Dependencies

The following command should be executed in the terminal of the system:

```
1  sudo apt-get install -y \  
2      python3-pip \  
3      build-essential \  
4      git \  
5      python3 \  
6      python3-dev \  
7      ffmpeg \  
8      libSDL2-dev \  
9      libSDL2-image-dev \  
10     libSDL2-mixer-dev \  
11     libSDL2-ttf-dev \  
12     libportmidi-dev \  
13     libswscale-dev \  
14     libavformat-dev \  
15     libavcodec-dev \  
16     zlib1g-dev  
17  pip3 install --user --upgrade Cython==0.29.33 virtualenv # the --  
    user should be removed if you do this in a venv
```

Code 2.1: Tools Command

The following packages have to be installed as well since they are required by the application. this requirements file has to be created before executing the command.

```
1 libbz2
2 python3
3 urllib3
4 pygments
5 idna
6 docutils
7 charset-normalizer
8 certifi
9 requests
10 Kivy-Garden
11 kivy
12 networkx
13 shapely
14 matplotlib
```

Code 2.2: requirements.txt

After creating the file, the following command should be executed

```
1 pip install -r requirements.txt
```

Code 2.3: Requirements installation

Buildozer [7]

Buildozer has to be installed with the following command:

```
1 pip3 install --user buildozer
```

Code 2.4: Buildozer Installation Command

Once Buildozer is installed on your system, The following command should be executed in the root folder of your source code, a buildozer.spec file appears on execution.

```
1 buildozer init
```

Code 2.5: Buildozer init

In the buildozer.spec settings have to be specified;

- **Basic Settings**

- title: Title of your application.

- `package.name`: Package name of your application.
- `package.domain`: Domain for your application package.
- `source.include_exts`: List of file extensions to include in the package :**py, png, jpg, kv, atlas, json, geojson**
- **Requirements** the packages found in the `requirements.txt` file have to be written in this field comma separated. Code 2.2
- **Application Icon** `icon`: Icon file (.png) for your application: **app_logo.png**
- **Orientation and Full-screen**
 - `orientation` : **portrait**
 - `fullscreen` : **0**

Make sure to add the following line at the end of your `/.bashrc` file, or it should be executed each time you open your terminal

```
1 export PATH=$PATH:~/local/bin/
```

Code 2.6: Path export

In order to build an Android version of the application the following command should be executed.

```
1 buildozer -v android debug
```

Code 2.7: APK building

If you have a slow computer. The first build will take time, as it will download the Android SDK [8], NDK [9], and others tools needed for the compilation. Don't worry, those files will be saved in a global directory and will be shared across the different project you'll manage with Buildozer.

Once finished, the file is created, and it can be found in the `bin/` directory. ready to be transferred to your device via USB or Bluetooth.

Upon clicking on the APK in your device, the installation procedure will start. Yet your phone may flag the app as possibly dangerous when you click on the icon to install it because it comes from an unknown source. You just have to bypass the warning and the installation will proceed.

Note: The app has already granted access to the storage of the phone in order to store its data

2.3 App Launch

Once the installation is successfully completed, your app will appear on your device's home screen and will be ready to use.

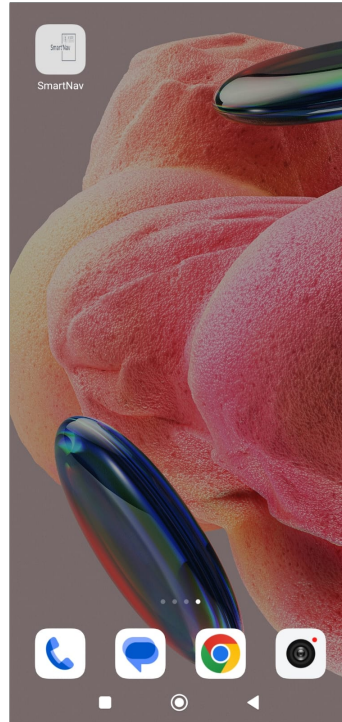


Figure 2.1: Application Icon

When the app comes with four screens during use:

1. Loading screen.
2. Mode-selection screen.
3. Destination-selection screen.
4. Map screen.

The UI Elements offered by the app or the device's back button can be used to navigate across these screens.

2.3.1 Loading screen

The first screen you see when you click on the app icon is called the loading screen.

In the worst situation, the loading screen during the app's first use could take up to six seconds to load the required assets and gather the required data. In other words, the first launch will prepare the needed data and save it for the next usage. Yet in the following instances, it will just take two seconds. Figure 2.2

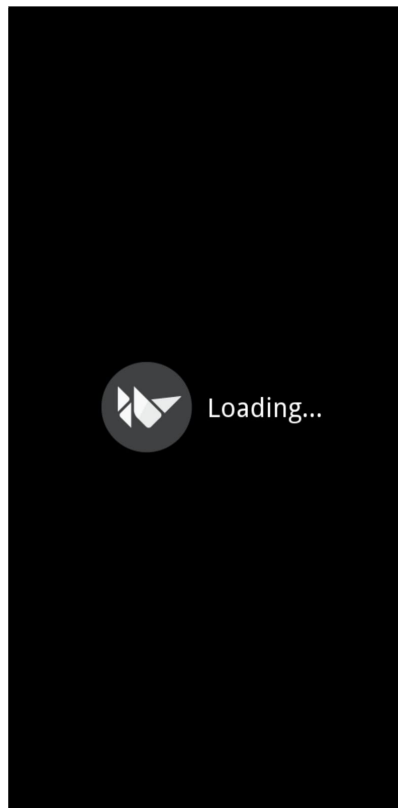


Figure 2.2: Loading Screen

2.3.2 Mode-selection screen

Depending on how they wish to use the application, users must press one of the three buttons on this screen.

1. **Normal.**
2. **Accessible.**
3. **Emergency.**

Modes	Meaning
<i>Normal</i>	To be used for users that are able to take both stairs and elevators.
<i>Accessible</i>	To be used for users that are limited to the stairs only.
<i>Emergency</i>	To be used in case of emergency, it leads to the closest EXIT.

Table 2.1: Application modes and their explanation

App behavior based on the selected mode:

- **Normal** : the app will provide paths to the destination based on the fact that the user can access both elevators and staircases.
- **Accessible** : the app will provide paths only through stairs if the destination and the current position floors are different.
- **Emergency** : the app will provide paths to the closest exits (existing only on floor 0), the passing from floor to floor can only be done through staircases for security reasons.

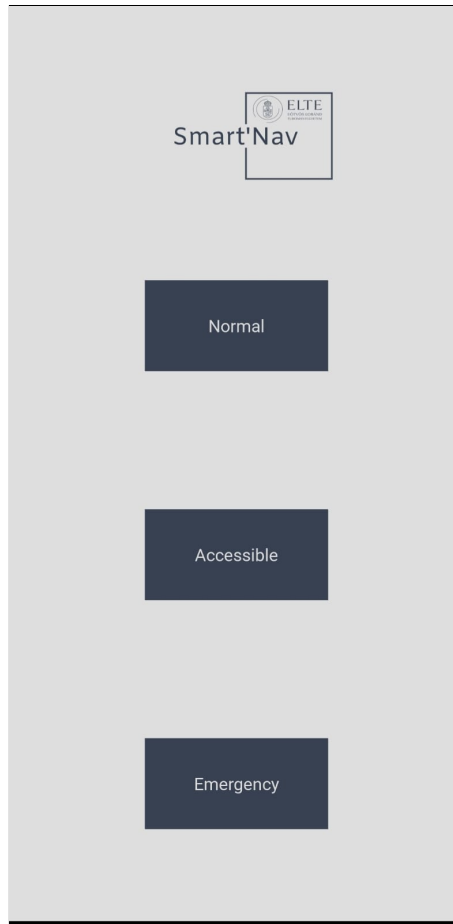


Figure 2.3: Mode selection screen

2.3.3 Destination-selection screen

Upon the selection of the mode, the selected value will be stored, and the app features will be generated based on that value as long as it's not changed manually. The app navigates to the Destination selection screen, where one of two screens will appear.

Destination-selection screen in Normal / Accessible mode

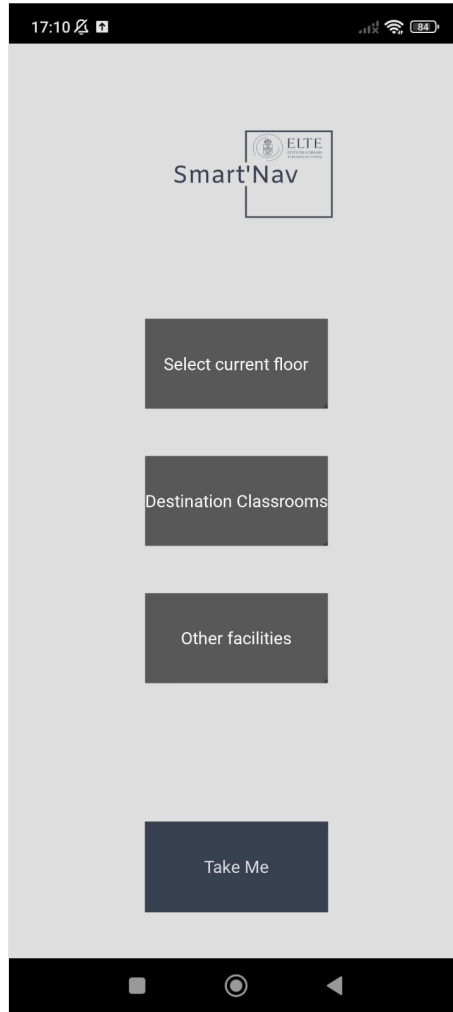


Figure 2.4: Destination selection screen Normal / Accessible mode

In case of selecting the Normal or the Accessible mode, this screen will appear as shown Figure 2.4. Three drop down menus with a submit button are available on the screen.

UI Element	Action
<i>Floors drop down</i>	Provides a list of the 9 floors of the building.
<i>Classrooms drop down</i>	Provides a list of the all the classrooms of the building.
<i>Other facilities drop down</i>	Provides a list of the facilities of the building : WC, Cafeteria and Libraries.
<i>Take me button</i>	navigates to the Map screen.

Table 2.2: UI Elements and their actions

Detailed explanation of each component:

- **Floors:** List of floors numbered from 00 (-1) to 7 where they serve to indicate the current position of the user floor wise.
- **Classrooms:** List of classrooms and offices of the whole building sorted in ascending order as follows : rooms starting with 0 then 00 then 1... **Note:** 00 is the floor -1 usually, but here it comes after 0 due to the **Lexicographic order** [10].
- **Other facilities:** List of the building's areas, including the men's and women's restrooms, the cafeteria (which is limited to the ground level), and the library (which has three locations: one on the ground floor and two on the first). If a facility has more than one location, the closest one will be selected as a destination.

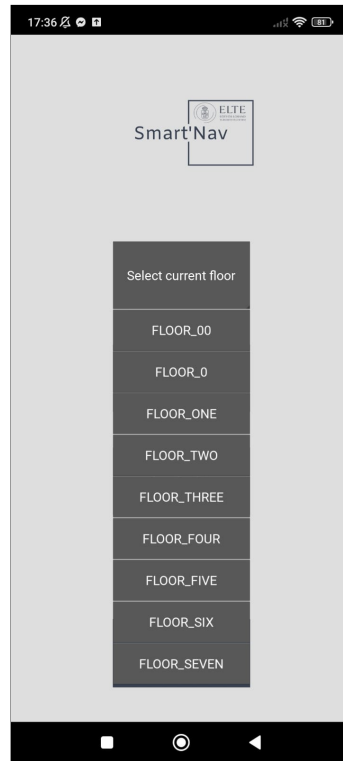


Figure 2.5: floors drop down expanded

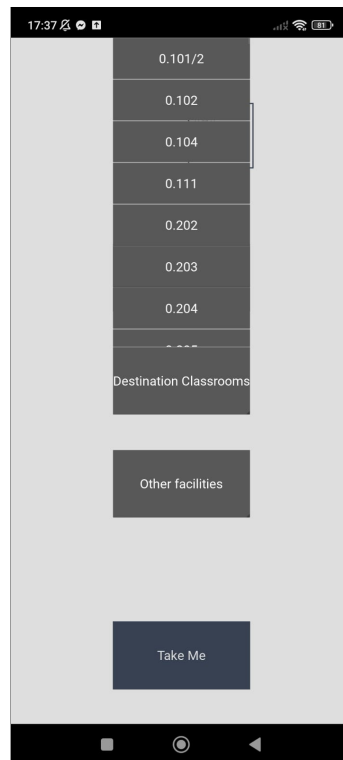


Figure 2.6: Classrooms drop down expanded

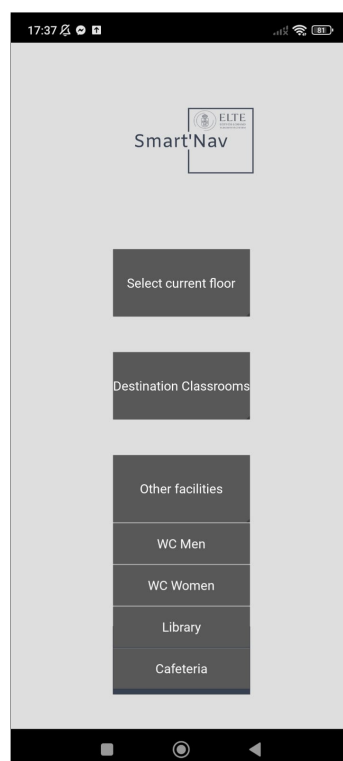


Figure 2.7: Other facilities drop down expanded

Using this screen is really simple and goes as follows:

1. **Select the current floor.**
2. **Pick a destination (Classroom or facility).**
3. **Press the Take Me button.**

NOTE: Choosing a facility after choosing the classroom will lead to setting the classrooms to default and pick the selected facility as the destination and vice versa.

Error-Indication Messages:

When the Take Me button is pressed, a form check will be initiated to verify that all conditions have been met. If successful, this will navigate to the Map screen; if unsuccessful, this will write an error message to the screen.

Select a floor Error: This error can occur when the current floor is not selected.

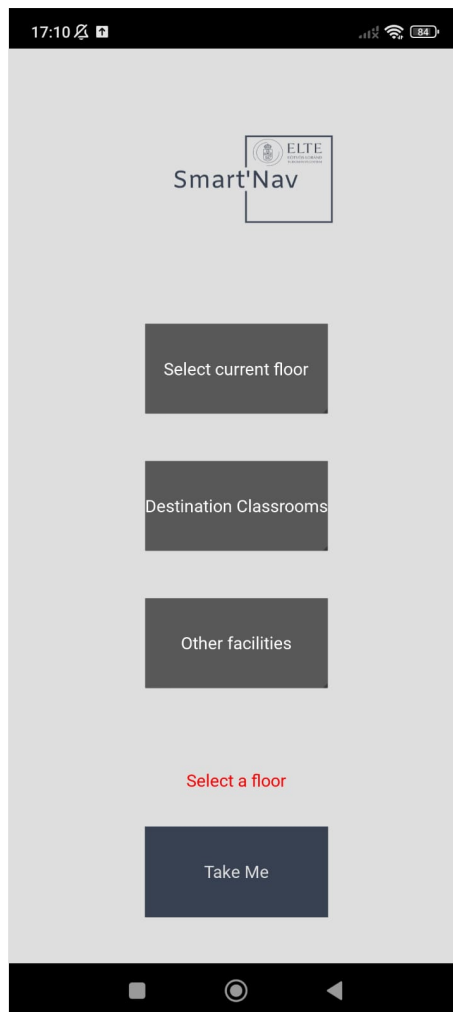


Figure 2.8: Select a floor Error

Select a destination Error: This error occurs when a neither a classroom nor a facility is selected as a destination.

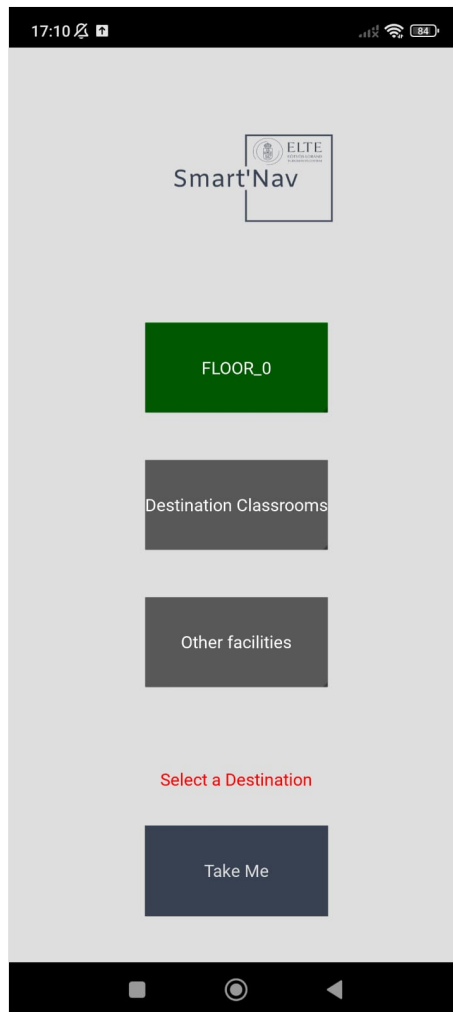


Figure 2.9: Select a destination Error

Color indications:

When a floor is selected, floors drop down can be colored in one of two ways: green if the floor has been implemented and its data is provided, or red with an error notice otherwise.

Note: Currently, the app has only been implemented for the first 3 floors. The rest of the floors would be implemented in case of the required data is provided.

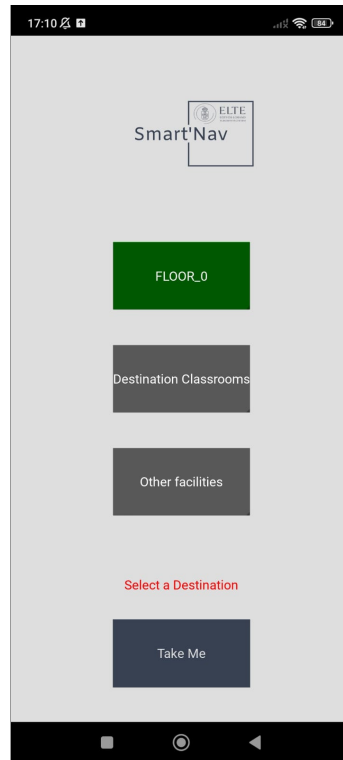


Figure 2.10: Green background floor

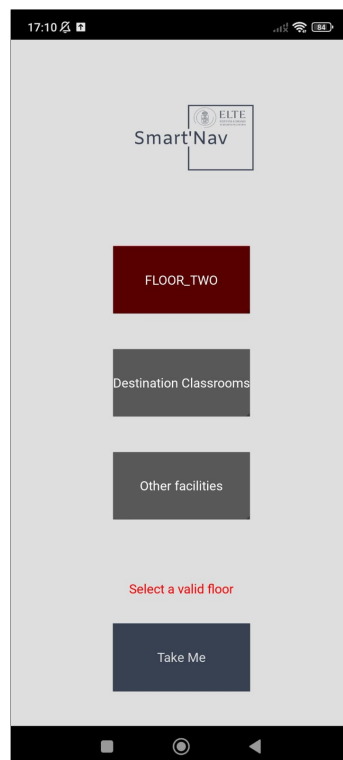


Figure 2.11: Red background floor

Destination-selection screen in Emergency mode

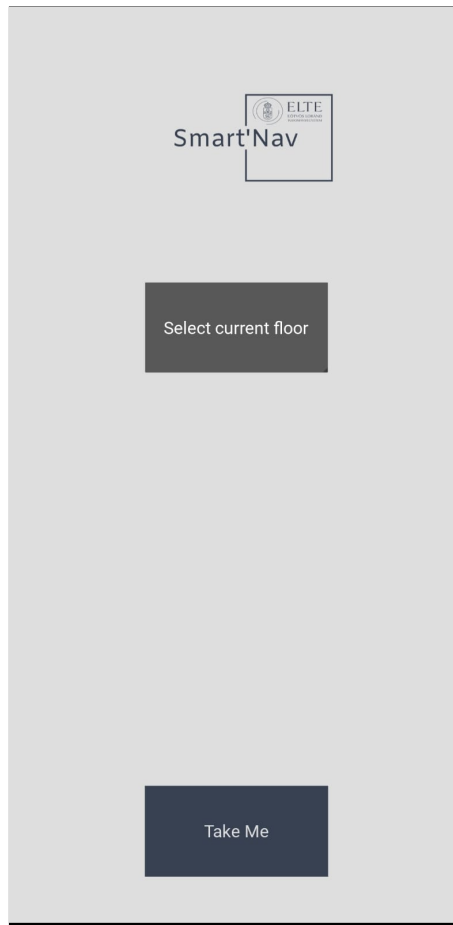


Figure 2.12: Destination selection screen Emergency mode

Upon choosing the Emergency mode on the Mode-selection screen, the application navigates to the Destination-selection screen, where only the floors drop down with the Take Me button are available. There is no need for the user to pick a destination, since it's automatically chosen to be the closest exit to the current position of the user. This screen also displays the error warnings for the floors drop down in addition to the color indicator.

2.3.4 Map screen

If passing the Destination-selection screen was successful. The users find themselves on the Map screen.

This screen is made mainly from three components

- **3D Map**
- **2D Mini-map**
- **Joystick**

3D Map:

The white-pink stripped walls that make up the 3D map shift in size, emerge, and vanish depending on where the user is in relation to the map. The 3D Map has the ability to present the walls in the application as they are, distance wise, in real life.

In other words, what a user would see in a real life situation, taking into consideration an average user height of 1.5 meter, is what the screen would display. For the purpose of keeping the view simple and the app as fluid as possible, distant walls cannot be seen.

Furthermore, the app's user has a field of vision that extends up to 60 degrees in both the horizontal and vertical directions.

The map doesn't show the building's classrooms / facilities labels, but it does show the doors, by mentioning a coloring depending on the nature of the door

- **Stair / Elevator** : The yellow color is used.
- **Exits** : The red color is used.
- **The rest of the doors** : The brown color is used.

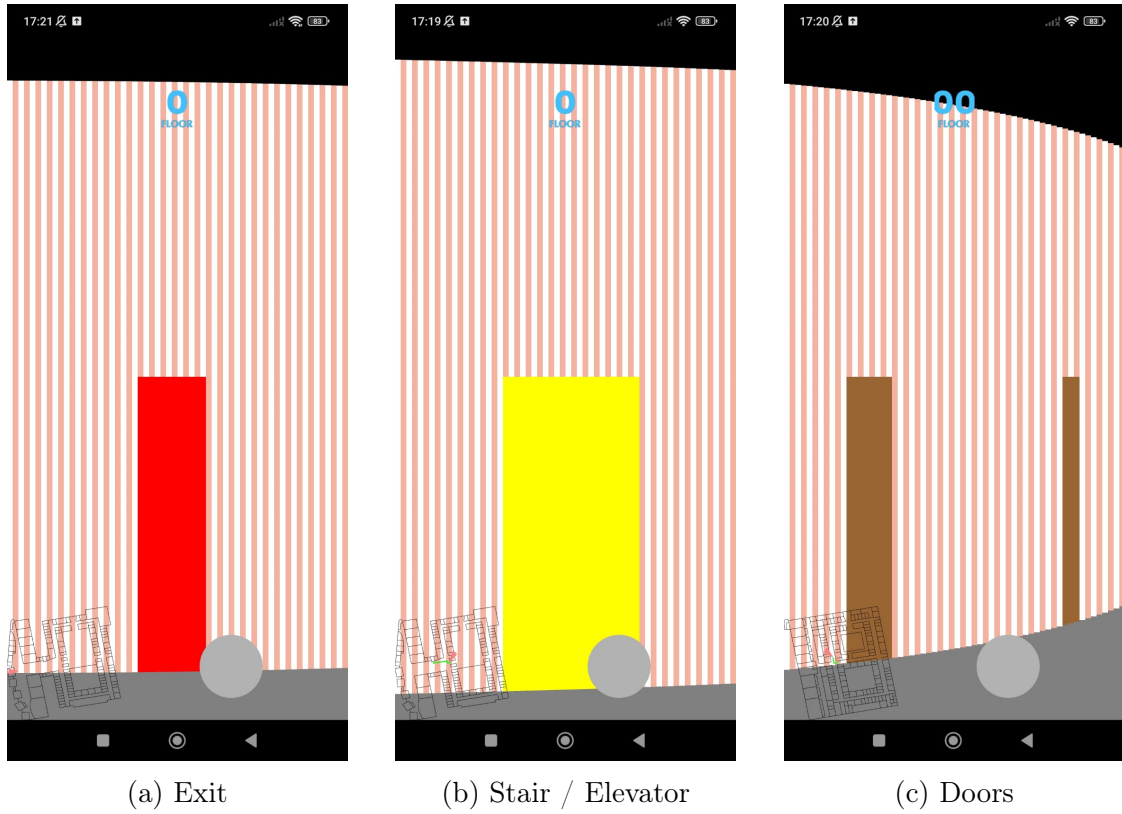


Figure 2.13: Doors Indications

Interacting with the 3D Map: The user can interact with the 3D map by swipe movements, where a swipe to the right causes the view to turn right or left otherwise. The angle of the turn and the length of the swipe on the screen are proportional, which assumes longer swipes = larger angle rotation. This action triggers the 2D mini-map user's field of view to rotate towards the opposite direction of the swipe.

2D Minimap:

The 2D Mini-map is located in the lower left corner of the screen. The primary objective of this function is to give the users additional information about their location within the building's limits.



Figure 2.14: 2D Minimap

This feature has also 3 main components:

- **Building's limits:** Lines forming the classrooms and the facilities
- **The User's position indicator:** Pink triangle presents the field of view of the user
- **Path:** Green line from the user's position to the target destination

2D Mini-map update: Any time the user interacts with the screen, the 2D Mini-map is updated in the way described below:

The location indicator of the user will move in the 2D map when they move, as will be covered in the next section.

When the user's field of vision changes, the position indicator will rotate in the direction of the selected field.

Path update: If the destination is on the same floor as the user, the path feature's green line will begin at the user's location indicator and continue to the destination. If there are two distinct floors, the path will begin at the user and lead to the elevator or stairs. Once there, it will update from the user's position to the destination after changing floors.



Figure 2.15: Path User - Connection Point



Figure 2.16: Path User - Destination

Note: The path will always be updated to give the shortest path, even if the user moves freely and exits the building. On higher floors, this might be viewed as a bug.

The path will calculate the shortest way by using the distances measured in meters. However, the approach thinks that there is no distance when it comes to stairs or elevators because it can't predict how quickly a person will move from one floor to another. Since it is a matter of time against the distance, both ways will have the same probability of happening when using the **NORMAL** mode.

Note: The back button press triggers the app to switch to the previous screen, which will reset the position of the user to the initial position and set all the fields to their initial state.

Joystick:

The users when accessing the Map screen will exist in the middle of the map where they need to reposition themselves based on their real position. The joystick is the feature that relates the position of the user to the real life position. The function of the joystick is helping the users reposition themselves inside the map by going in the 4 directions.

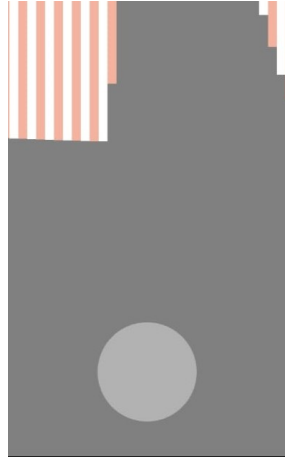


Figure 2.17: Joystick

The app is implemented in a way that distinguishes a joystick movement from a change of a field of view by checking if a touch lays within the lower 1/3 of the screen then it engages the joystick, otherwise any horizontal movement within the rest 2/3 will be interpreted as a view swipe (discussed previously)

The joystick functioning: When the joystick is moved in a certain direction, the screen updates to reflect the user's movement in that direction. As a result, the user's location will continuously be updated as long as the joystick is held away from its starting point.

The map stops updating, and the joystick returns to its starting location when the joystick is released. However, if the user is too close to a wall, the movement will be suspended towards that wall due to the collision detection feature that prevents the user from walking into walls.

Other Map screen features: The application gives the user an indication visually when they are in the map screen for the ease of use. The label at the top of the screen, known as the floor indication, shows the user which floor they are currently on.

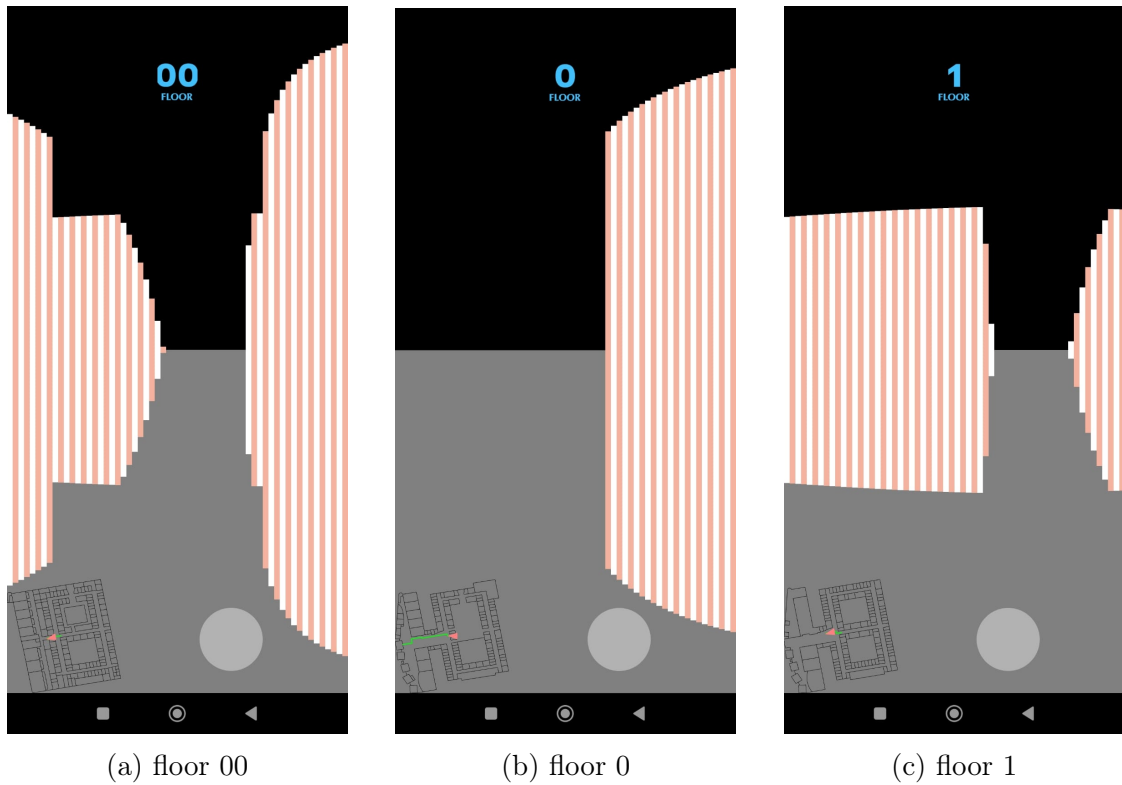


Figure 2.18: Floors indication.

Furthermore, a popup screen will open in the following two cases:

- **Arriving at an elevator or staircase:** The popup will appear labeled as Take Elevator / stairs to the [destination floor] and will show the new floor

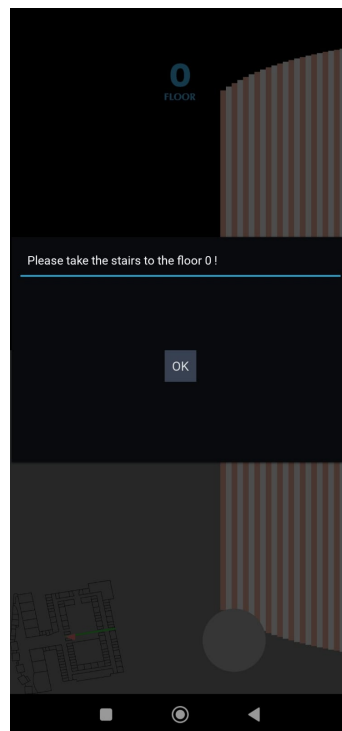


Figure 2.19: Connection point popup

- **Arriving at the chosen destination:** The popup will appear labeled as Congrats, you reached your destination!

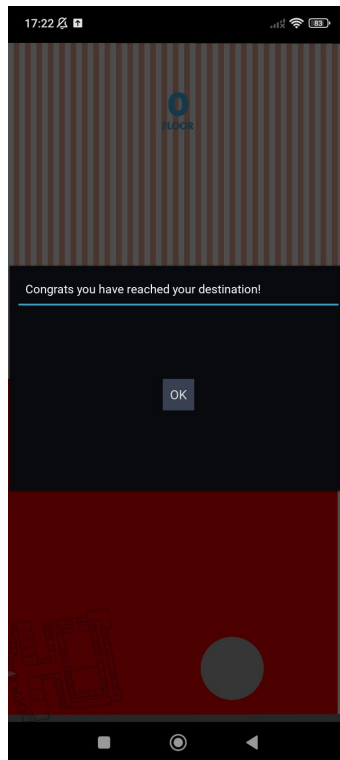


Figure 2.20: Reached destination popup

The OK button will appear within the popup, upon press, it closes the popup and stays on the Map screen. In this case, the destination will remain the same. If the users intend to change any of the setting they have already set, they should go back to the previous screens.

Conclusion:

Although SmartNav isn't the best app for indoor navigation, it attempts to simplify the intricate layout of the building by offering both 2D and 3D maps that complement each other to complete the task.

Chapter 3

Developer documentation

SmartNav attempts to create a realistic perspective of the faculty's layout and simplify navigation, the application attempts to turn the 2D maps of the building into a 3D display, and gives the ability of interaction with the UI to the user.

3.1 Design

3.1.1 Introduction

The application's primary design concepts are **performance**, which highlights each component's ability to deliver a rapid, responsive experience, and **simplicity**, which concentrates on important aspects and eliminates unnecessary, complex features. Furthermore, the application considers user convenience by offering a mobile app that lets users have a portable navigation system which they may use in real-world scenarios. This justifies the decision to make the application a mobile one. Additionally, users can operate the program in offline mode, which allows them to locate their way without an online connection.

3.1.2 System Architecture

The Model View Controller architecture [11] (**MVC**) used in the construction of the software makes data flow easier to manage while it runs. The Controller has control over the situation with this technique: The data is retrieved by this last from the model, which acts as storage, and the business logic package, and it is then sent to the View package, which displays the data that has been retrieved, and vice versa.

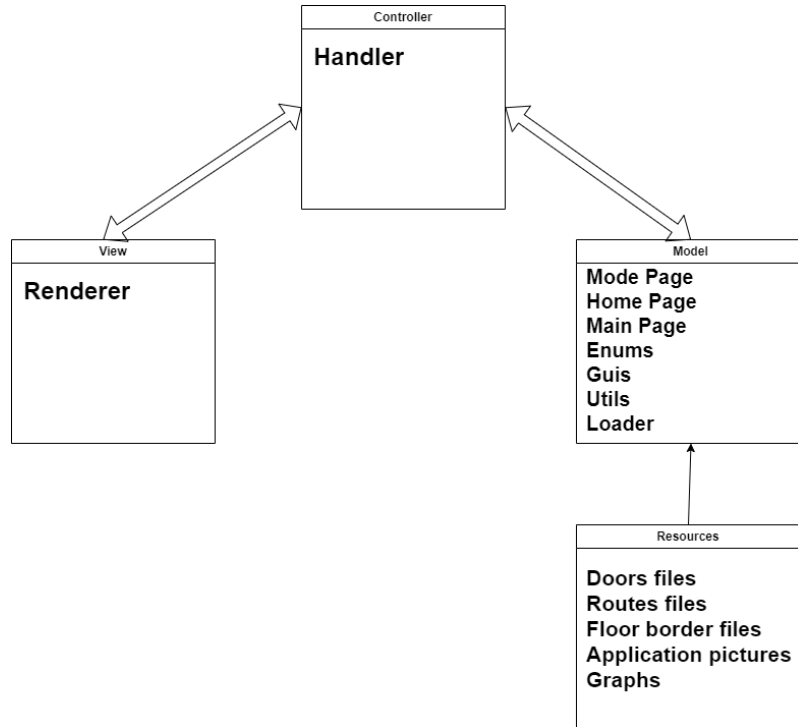


Figure 3.1: Packages Diagram

Details description of the packages:

- **Controller** : The Handler Class manages event listening and facilitates the connection between the view and the model. When an action is initiated from the view, it is communicated to the handler, which then determines and engages the appropriate part of the model.
- **View** :The Renderer class is responsible for displaying the UI elements to the user and forwarding any triggered events to the handler.
- **Resources** : Acts like the database of the application where all the needed data files and Textures are stored.
- **Model** : The model package contains the core classes of the application, including the main screens such as the mode page, home page, and main page. The most important class among the three is the main page, also known as the

map page. This class contains the logic that interconnects the Student, Map, Doors, Path, and Routes classes.

3.1.3 Application Layout

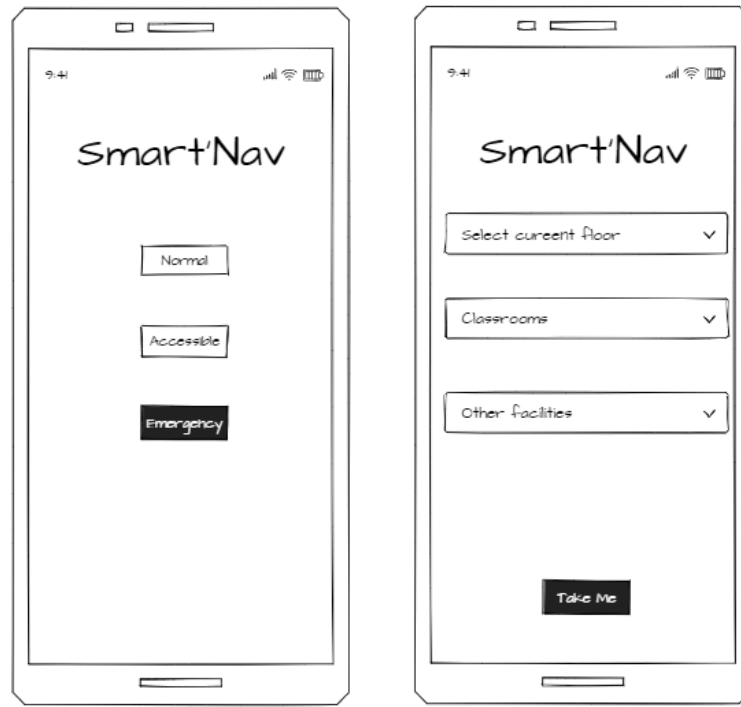
In order to keep the Application straightforward and simple. The design intends to minimize the number of screens and UI elements. In other words, the application accomplishes its primary objective in the most basic way possible. Furthermore, It uses labeled widgets as UI components to reduce the display of indications, to avoid screen congestion.

The SmartNav App tends to have one main objective to achieve, which is providing paths for destinations requested by the users.

Yet in real life situations, the users may have physical constraints that the app needs to adapt to by having **Modes**.

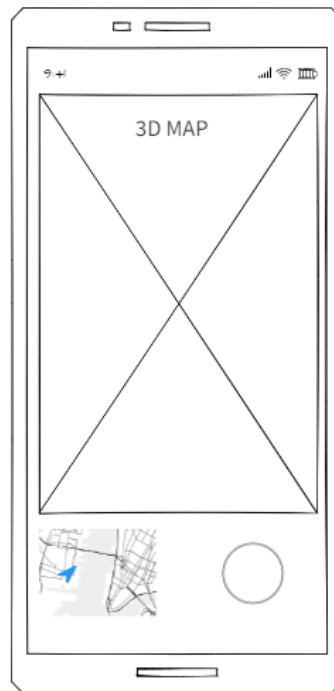
By supporting various user types, these modes make sure the program has the requirements for any use scenario. As a result, the idea of employing screens was created to allow users to select their specific needs or constraints upfront, so they continue using the app in that specific way. Moreover, the screens are ordered as following:

1. **Modes screen** allows users to specify the mode upfront.
2. **Home screen** The Destination selection screen allows users to change their destination while keeping their selected mode unchanged.
3. **Main screen** For navigation purposes, to remove all the settings widgets and focuses on delivering the navigation service.



(a) Modes page

(b) Home page



(c) Main page

Figure 3.2: Wireframes

UI components

The choice of the UI elements is a factor that impacts the user-friendliness of the application. In other words, the less screen taps, the faster the goal of the app is delivered. That justifies the choice of the following elements.

Widgets

- **Buttons:** Provide immediate action with a single tap.
- **Drop-downs:** Help gather the long lists in one slot with the ability to scroll through them. Avoids the typing that can cause issues related to typos.
- **Joystick:** Enhances the navigation and provides space efficiency on the display.
- **Pop-Ups:** Pop-ups are used as checkpoints within the application. They block the app while open, giving users enough time to respond to the information or actions required. Plus, they are used to signal that a process has ended, ensuring users are aware of completion and can take any necessary follow-up actions.

Touch interactions: In the Main screen, where the 3D map is, the App allows the users to use touch gestures to avoid multi-taps and to replace the add of widgets on the screen by simple touches

The 2D - 3D Combination Although SmartNav is essentially a 3D application, it is necessary to have a 2D map in order to prevent confusion caused by unclear 3D models. Its function is to support the main map by offering features that the main map misses due to implementation challenges.

3.1.4 Data Sources

Faculty's floor wall maps:

The faculty provides professors and students with wall maps at each floor that indicates where the person is standing and shows a layout of the whole floor with the rooms and facilities labeled. These maps can be found on the faculty's website in picture format.

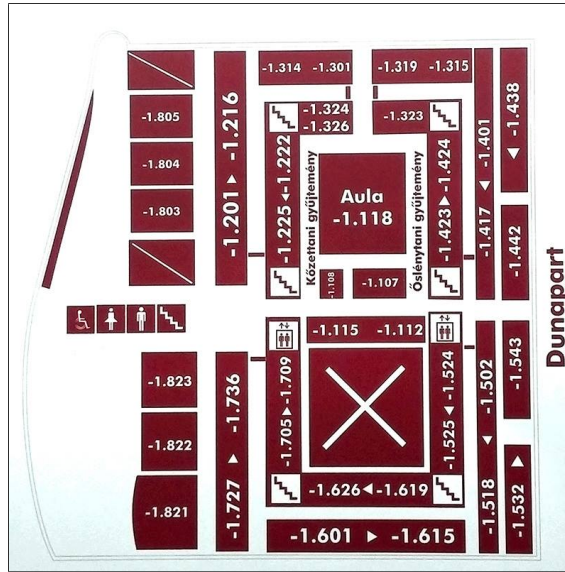


Figure 3.3: Faculty's floor 1 wall map

The original approach involved subjecting the pictures to an image processing procedure, resulting in them being converted to black and white bitmap format. This allows them to be stored in binary format, which facilitates the detection of walkable areas within the building limits using the flood fill algorithm. However, this type of data can cause issues in accurately locating the components of the building.

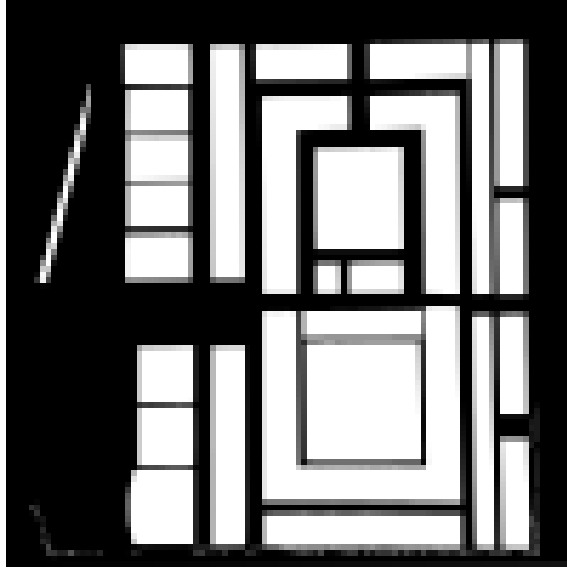


Figure 3.4: Faculty maps processing

QGIS

QGIS [12] (Quantum Geographic Information System) is a free, open-source software that allows users to create, edit, visualize, analyze, and publish geospatial information.

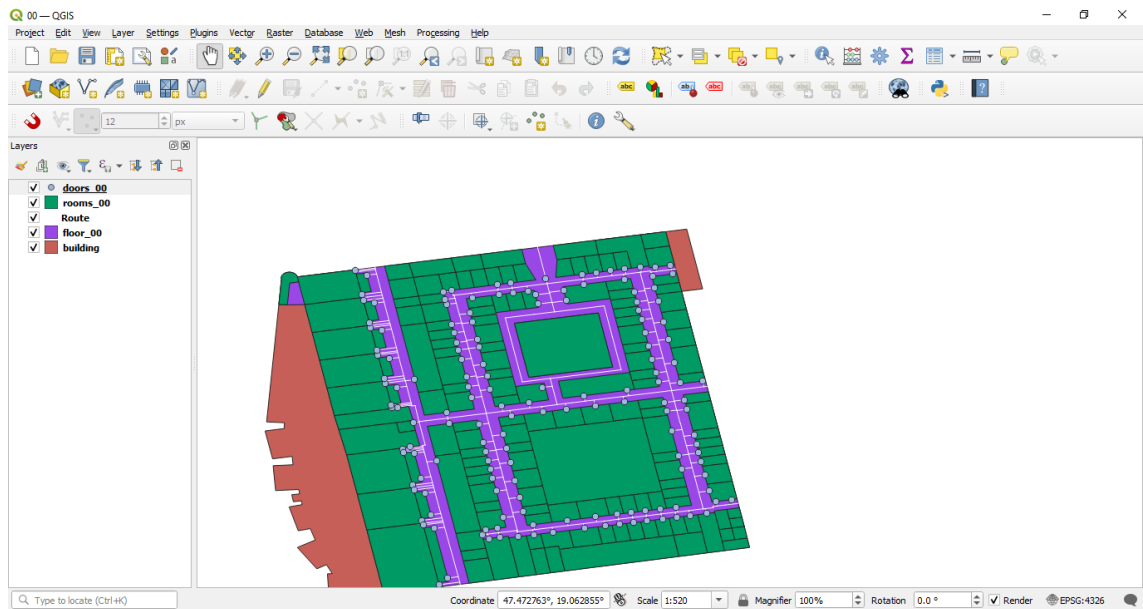
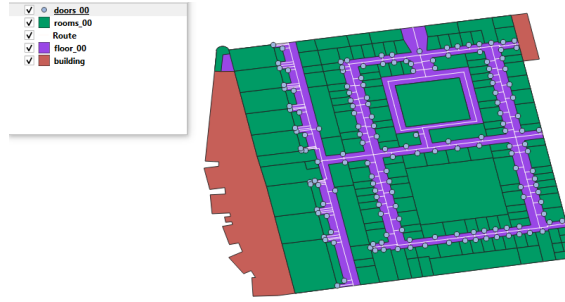


Figure 3.5: QGIS interface

With the help of **The Department of Cartography and Geoinformatics, Eötvös Loránd University.** [13] QGIS project folders for the first 3 floors were put up. These projects gather all information related to these floors of the building:

- Doors, Elevators and Staircases
- Routes
- Building limits

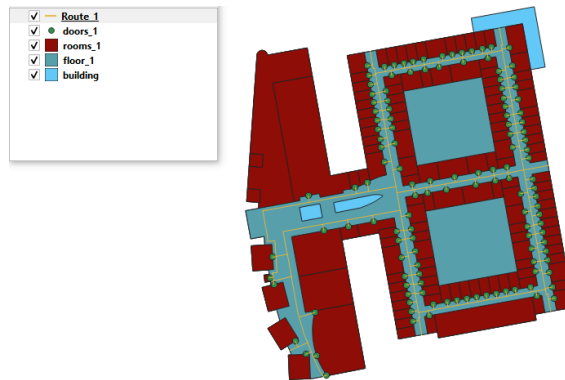
The QGIS projects, after processing, provide data as geojson [14] files including the observed following shapes with global longitude and latitude coordinate system. The use of the global coordinate system with an accuracy of $10E-15$ gives the app the realistic aspect, especially when it comes to calculating distances.



(a) Floor 00



(b) Floor 0



(c) Floor 1

Figure 3.6: QGIS floor representation

Linking the floor components:

The floor elements can be mapped to a weighted graph with the following design idea:

The doors of each classroom and facility will be the nodes of the graph, and the routes should represent the edges having the distances in meters as the edges' weights. This graph will serve for the determination of the shortest path. Moreover, the floors will be interconnected using the elevators and the stairs as the edges between each floor's graph depending on the mode selected; The General graph will remove edges of elevators if the Emergency mode is selected. For the Accessible mode, the stairs' edges will be omitted.

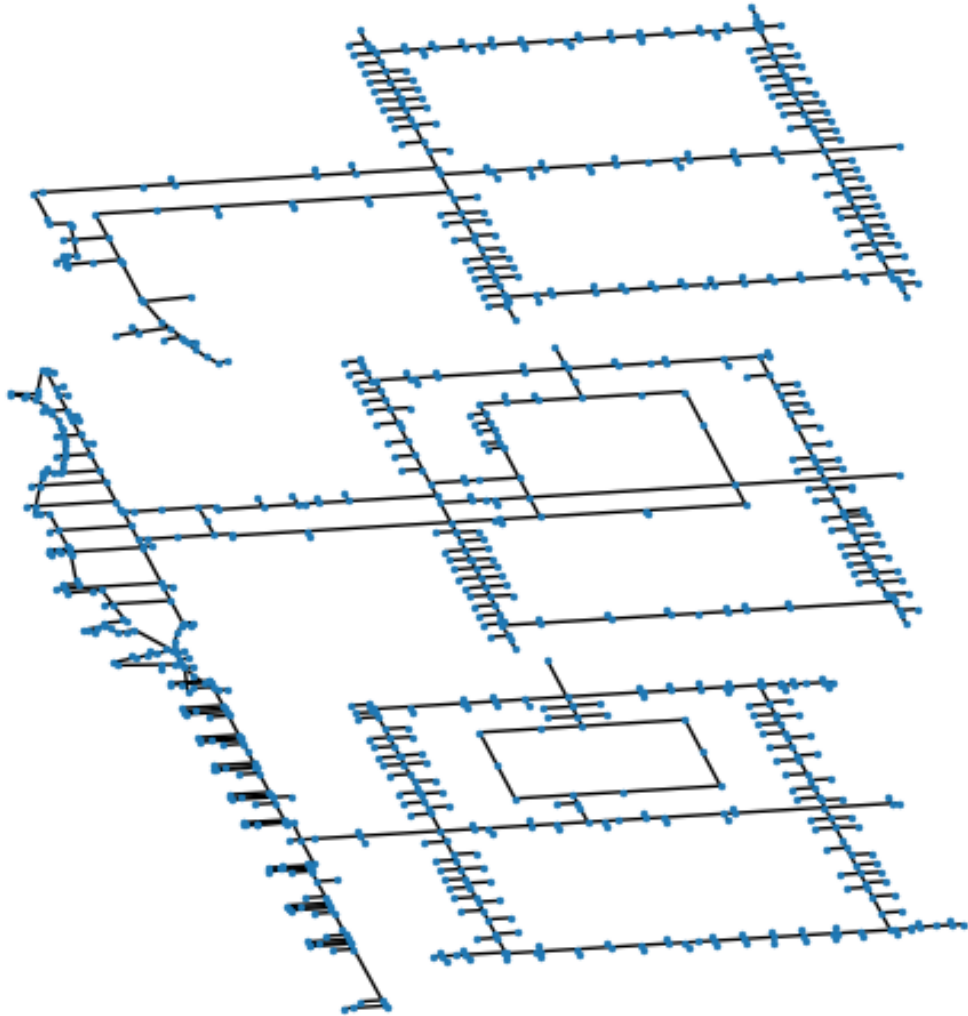


Figure 3.7: Graphs of floors

3.1.5 3D Modeling

Introduction

The approaches for the 3D modeling are many, but each comes with a cost especially when we mention mobile devices, since they have less processing power than desktop computers, which can make rendering complex 3D models slow and inefficient, and can also drain their battery quickly. Besides the memory that is limited for both RAM and storage, there is the constraint of low-ends GPUs.

For this app, the choice of the traditional **Ray Casting** [15] technique was a proper one due to its simplicity and efficiency of implementation and being suitable for real time applications with its lightweight rendering.

Tackling the issues

Moving from a point and few polygons to 3D representation can be challenging without the right steps. The idea mainly came from a game called [16], where they use ray casting techniques to drift from 2D to 3D.



Figure 3.8: Doom Game

This game focused mostly on three main concepts:

- 2D layout and Textures
- Height Variations
- Ray Casting

For the practical use of these concepts in terms of SmartNav, these concepts were designed phase as follows:

Ray Casting

The ray casting algorithm utilizes basic mathematical principles to measure distances between each line of the surrounding polygons and the user, represented as a point on the map.

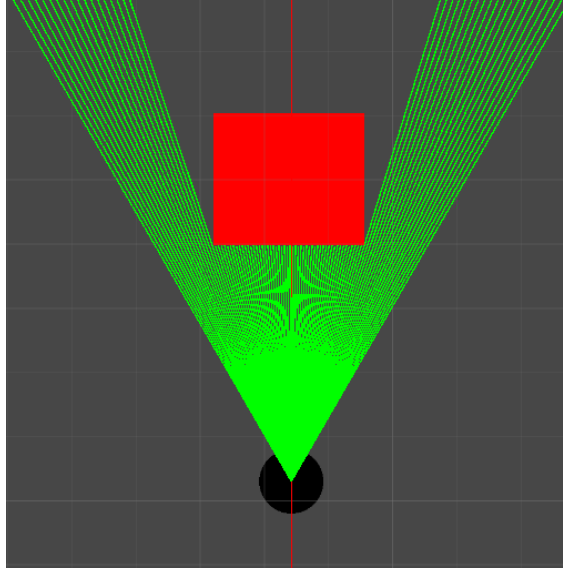


Figure 3.9: Ray casting example

To display the 3D map view on the screen, the user casts 60 rays at 60° angles. Each ray returns information from its ray casting algorithm, identifying the closest point in that direction.

Occlusion is implemented in this algorithm, ensuring that only the nearest wall is rendered and others are avoided.

If a wall is too distant (its distance in pixels exceeds half the height of the screen), it is not rendered. The apparent height of each visible wall is calculated based on its distance from the user, ensuring that distant walls appear shorter on the screen.

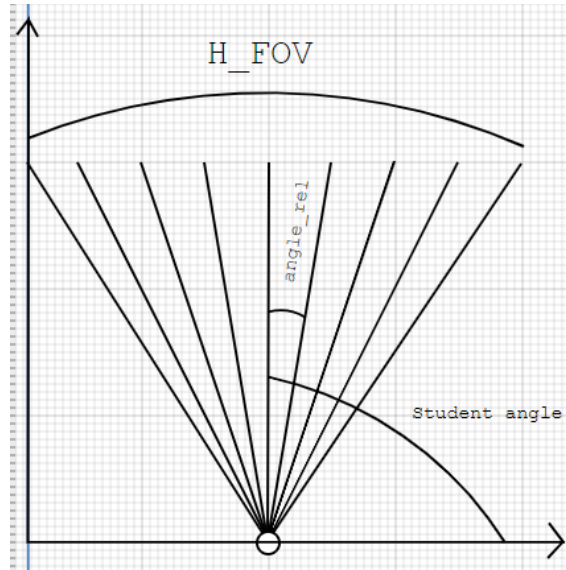


Figure 3.10: The user's Horizontal FOV

the Map creation and 2D Textures: The 2D map layout designed in a way that the classrooms as polygons construct the map of the application, and primitive colors will be the main factor of textures.

For the sake of consistency in calculating distances and projections, multiple coordinate systems were designed as follows :

- World Coordinates : Longitude and Latitude, which are the coordinates provided by the geojson files.
- Mini Map Coordinates : relative to the 2D Mini map.
- Building Coordinates : which are normalized coordinates; South-West 0 and North-East 1
- User Coordinates : coordinates that are relative to the height
- Screen Coordinates : Pixels

The height variation: The objects' heights will adjust proportionally to the distances in order to provide the users with a 3D experience.

- **"USER HEIGHT"** : 1.5 Meters
- **"WALL HEIGHT"** : 3 Meters
- **"V FOV"** : 60°

This design helps in adding the realism to the map by having the user sight at the half of the wall and as well the way the users see the wall will depend on how far they are from it (explained in the ray casting subsection).

However, the doors for classrooms, elevators / stairs and exits will be rendered with the size of half a wall and colored with their specific colors.

Note: Although the user has a vertical field of view, the ray casting only takes place on the 2D map.

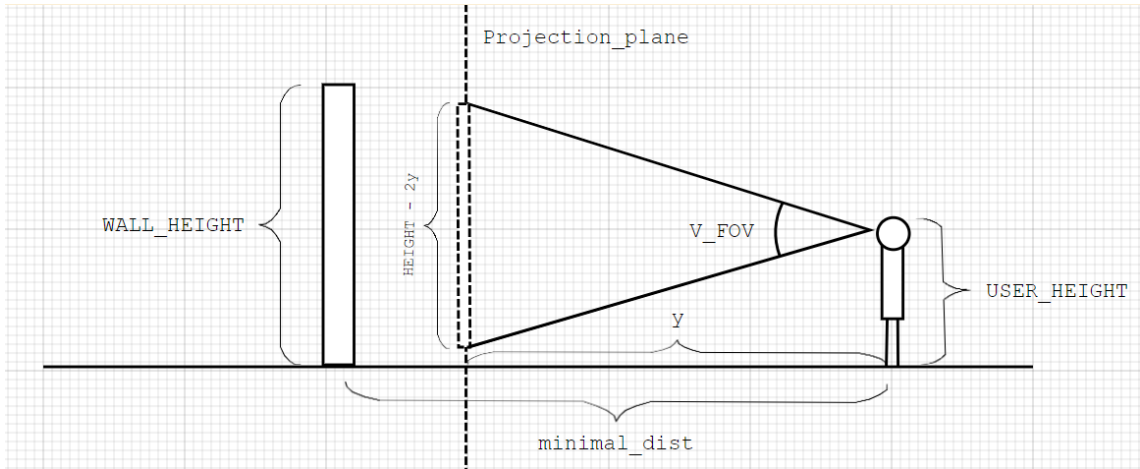


Figure 3.11: Height and Vertical Field of view

3.1.6 2D Modeling

The mini map is continuously displayed on the main screen to provide supplementary information to the 3D map and compensate for missing features, such as showing the path. This path is drawn on the 2D map to indicate the route the user should take to reach their destination, updating dynamically as the user moves. The path is only drawn on walk-able areas.

User Representation

The user is represented on the map by their horizontal field of view. Rays emitted from the user's current position will be drawn, with the origin of these rays corresponding to the user's location within the map. However, the rays are visually represented to extend only up to 10 meters for aesthetic purposes, although the actual ray casting calculations go to infinity

User Movement

The user can move freely within the map but is restricted to walk-able areas, such as corridors and open spaces.

A collision detection algorithm prevents the user from passing through walls. This concept is inspired by Taxicab geometry [17], which measures distance as the sum of the absolute differences of coordinates, rather than the Euclidean distance. In this

application, distances are calculated as the total weight of the edges between two points.

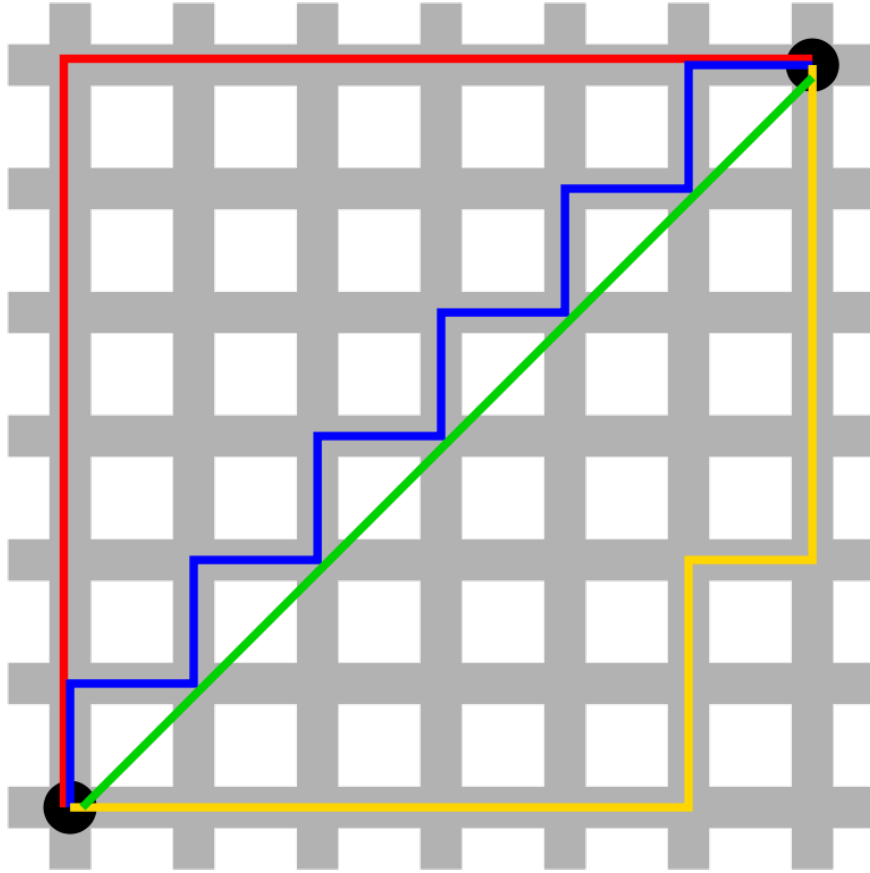


Figure 3.12: Taxicab geometry

3.1.7 Shortcomings

During the application’s design phase, several questions arose regarding its features and how to achieve them while maintaining high performance.

3D Path Issue

A major issue was the inability to implement 3D path drawing due to its complexity. To work around this, the path is drawn in 2D. Yet this solution had its own problems since the ray casting algorithm failed to detect intersections when the user was collinear with the path, causing the path not to display. To fix this, the idea was breaking the path into segments and rotating them 90 degrees for ray casting, but the path still did not render well. This remains an area needing further improvement.

Another approach was to use arrows in the 3D map to show the direction to go. However, this method faced challenges with alignment, requiring angle calculations to accurately determine the direction between segments, the user's direction, and the nearest path segment which needs an absolute angle to be determined.

Data Inaccuracies and deficiencies

Despite the projects providing the application with necessary information regarding the building's elements and measurements, it still suffers from inaccuracies, particularly in missing elements like the large staircase or the elevators. This affects the application's efficiency in determining the shortest path in real-life scenarios with 100% precision.

Furthermore, the projects provided data only for the first three floors of the building. To address this issue, adding the required data to the resources package can resolve the issue without requiring changes to the core code of the application.

Poor texture

To ensure the application maintains high performance, SmartNav might simplify textures by using primitive colors as indicators for elements on the 3D map. However, these simplifications can be replaced with real textures through the implementation of ray tracing for lighting effects and vertical ray casting for accurate representation of ceilings and floors.

3.2 Implementation

3.2.1 Introduction

SmartNav is developed using Python [3] and Kivy [18]. Python is chosen for its readability and extensive library ecosystem, which enable quick development and excellent functionality.

Kivy provides tools to create apps with touch features and multimedia support. This section explains how Python and Kivy were used to create a 3D navigation app that's easy to use on mobile android devices. It also shows how these tools were leveraged to optimize the user interface for natural interactions.

Moreover, Kivy provides capabilities for drawing on the screen's canvas, allowing to implement custom drawing and interactive elements with ease and flexibility that are tailored precisely to the app's objective needs.

3.2.2 Data Processing

Data extraction

The following steps need to be followed in order to extract the geojson files from the QGIS project:

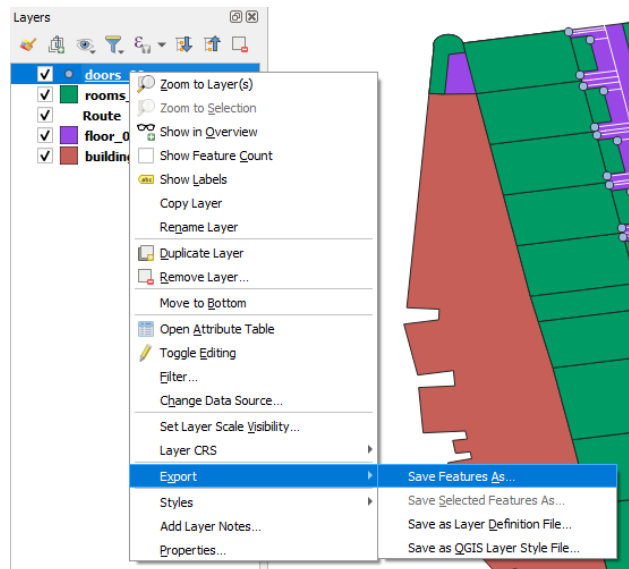


Figure 3.13: QGIS layer to export

Upon completion of the previous step, the following window will appear where the coordinate reference system has to be selected as the global latitude / longitude.

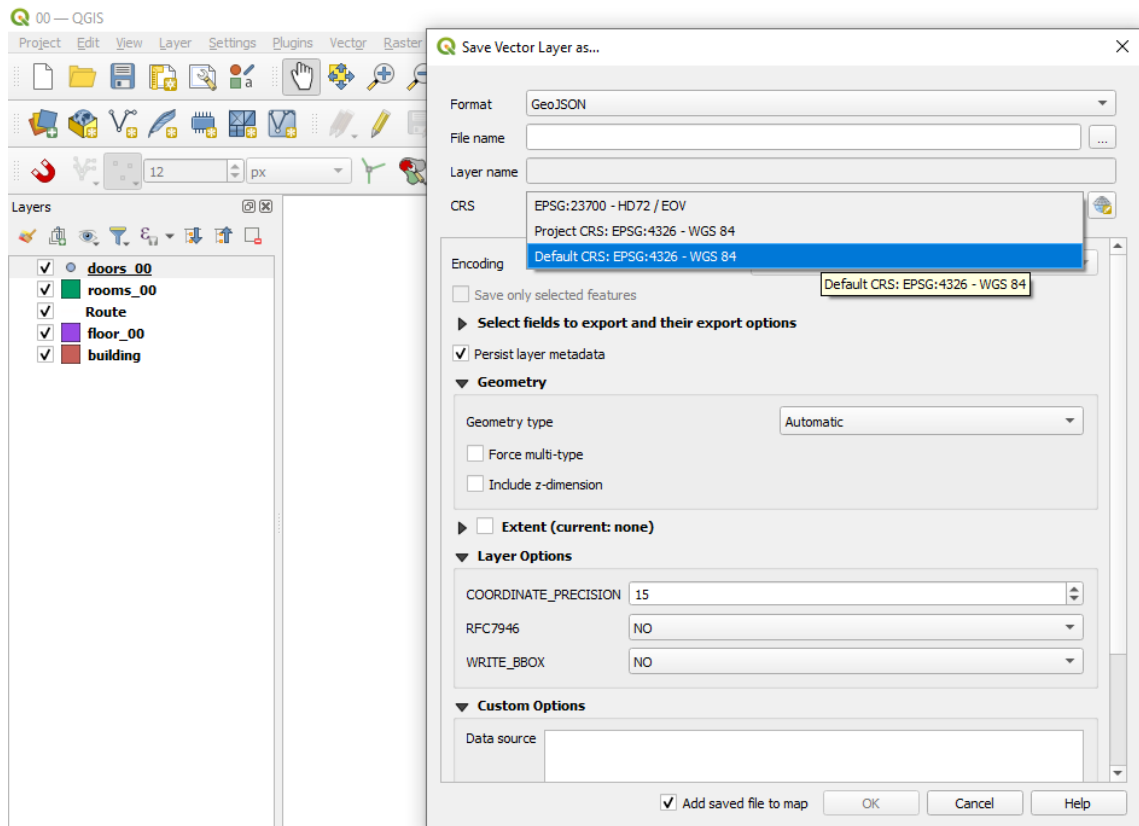


Figure 3.14: Selecting the Coordinate system

Finally, the precision of the coordinates has to be precised in this box where 15 has been selected for all the data of the application:



Figure 3.15: Precision selection

The final result of the extraction will be a geojson as the example below:

```
{
  "type": "FeatureCollection",
  "name": "doors-1",
  "crs": {
    "type": "name",
    "properties": {
      "name": "urn:ogc:def:crs:OGC:1.3:CRS84"
    }
  },
  "features": [
    {
      "type": "Feature",
      "properties": {
        "type": "other",
        "id": "00.403",
        "name": "Hullad  kt  rol  ",
        "geometry": {
          "type": "Point",
          "coordinates": [
            19.063009392072509, 47.472784858244928
          ]
        }
      }
    },
    {
      "type": "Feature",
      "properties": {
        "type": "other",
        "id": "00.402",
        "name": "  llati tetem h  tt  k",
        "geometry": {
          "type": "Point",
          "coordinates": [
            19.062962588597696, 47.472779136245293
          ]
        }
      }
    },
    {
      "type": "Feature",
      "properties": {
        "type": "other",
        "id": "00.319",
        "name": "G  p  szet szell  z   g  ph  z",
        "geometry": {
          "type": "Point",
          "coordinates": [
            19.062914195658898, 47.472778085086397
          ]
        }
      }
    },
    {
      "type": "Feature",
      "properties": {
        "type": "other",
        "id": "00.318",
        "name": "Steriliz  l  ",
        "geometry": {
          "type": "Point",
          "coordinates": [
            19.062865167676854, 47.472772733197218
          ]
        }
      }
    },
    {
      "type": "Feature",
      "properties": {
        "type": "other",
        "id": "00.317",
        "name": "Kronobil  gia",
        "geometry": {
          "type": "Point",
          "coordinates": [
            19.06282166491907, 47.472767415187789
          ]
        }
      }
    },
    {
      "type": "Feature",
      "properties": {
        "type": "lab",
        "id": "00.316",
        "name": "Laborfelel  s",
        "geometry": {
          "type": "Point",
          "coordinates": [
            19.062776521987423, 47.472761445936284
          ]
        }
      }
    },
    {
      "type": "Feature",
      "properties": {
        "type": "Office",
        "id": "00.315",
        "name": "E  gtv  s Informatikai K  r Csoportszoba",
        "geometry": {
          "type": "Point",
          "coordinates": [
            19.062732823572481, 47.472756117438557
          ]
        }
      }
    },
    {
      "type": "Feature",
      "properties": {
        "type": "other",
        "id": "-1.330/A",
        "name": "TTK Tank  gnyv   s Jegyzetbolt",
        "geometry": {
          "type": "Point",
          "coordinates": [
            19.062617631101197, 47.472741960698528
          ]
        }
      }
    },
    {
      "type": "Feature",
      "properties": {
        "type": "Storage",
        "id": "00.323",
        "name": "Rakt  r",
        "geometry": {
          "type": "Point",
          "coordinates": [
            19.062738321099204, 47.472724596618619
          ]
        }
      }
    },
    {
      "type": "Feature",
      "properties": {
        "type": "Stairway",
        "id": "00.320",
        "name": "Stairway",
        "geometry": {
          "type": "Point",
          "coordinates": [
            19.062866798916072, 47.472740110200384
          ]
        }
      }
    },
    {
      "type": "Feature",
      "properties": {
        "type": "Storage",
        "id": "00.409/a",
        "name": "TEF rakt  r 1.",
        "geometry": {
          "type": "Point",
          "coordinates": [
            19.063018425988712, 47.472756545585057
          ]
        }
      }
    },
    {
      "type": "Feature",
      "properties": {
        "type": "other",
        "id": "00.409",
        "name": "Es  ztet  ",
        "geometry": {
          "type": "Point",
          "coordinates": [
            19.062954899672622, 47.472748583932038
          ]
        }
      }
    },
    {
      "type": "Feature",
      "properties": {
        "type": "other",
        "id": "00.408",
        "name": "Szell  z   g  ph  z",
        "geometry": {
          "type": "Point",
          "coordinates": [
            19.062960744607757, 47.472714329934568
          ]
        }
      }
    },
    {
      "type": "Feature",
      "properties": {
        "type": "other",
        "id": "00.430",
        "name": "G  p  szet",
        "geometry": {
          "type": "Point",
          "coordinates": [
            19.06289653486445, 47.472780027282083
          ]
        }
      }
    },
    {
      "type": "Feature",
      "properties": {
        "type": "lab",
        "id": "00.410",
        "name": "PC-labor",
        "geometry": {
          "type": "Point",
          "coordinates": [
            19.06297283984652, 47.472670549733415
          ]
        }
      }
    },
    {
      "type": "Feature",
      "properties": {
        "type": "other",
        "id": "00.429",
        "name": "nothing",
        "geometry": {
          "type": "Point",
          "coordinates": [
            19.062904049155417, 47.472669167950073
          ]
        }
      }
    }
  ]
}
```

Figure 3.16: Example Geojson

Graph generating

NetworkX

To manage and represent the indoor navigation paths, the NetworkX [19] library was utilized. NetworkX allowed us to create detailed graphs that model the navigable areas within the building. These graphs facilitated efficient pathfinding and route optimization by representing navigable paths as nodes and edges. The choice of networkX was made upon the following key factors:

- **Easy Graph Representation:** easy generation and modification of graphs. It allows viewing the graphs by using the Matplotlib library.
- **Built-In Pathfinding:** NetworkX has many built-in algorithms for finding the shortest paths and optimizing routes.
- **Real-Time Updates:** the graph gets updated real-time, that helps adjust to the moving user of the app.
- **Easy Python integration:** works smoothly with your existing app code, making it easy to manage and use navigation graphs.

Routes Issues

Due to the high accuracy of the coordinates in the route files, generating the graphs presented issues. The lines appeared to intersect, but on a microscale, they did not actually touch. This led to the idea of implementing a custom algorithm for graph generation using the **Shapely** library.

The choice of Shapely [20] came from the fact that The geojson files provide geometric objects like MultiPoint, MultiLineString, MultiPolygon. And since Shapely provides a rich set of geometric operations, including: Union, intersection, and difference of shapes, that can be used on these geometric objects, it makes it the right tool for this mission. Moreover, the routes posed an issue with large segments, which in turn affected the accuracy of shorter paths. This is because intersections of routes were not considered as nodes in this case.

Graph Generation steps

Read the Geojson file and store its components.

```
1 def get_vertices_and_edges(self):
2     multilinestring_coords = []
3     for feature in self.routes["features"]:
4         if feature["geometry"]["type"] == "MultiLineString":
5             multilinestring_coords.extend(feature["geometry"]["
6                 coordinates"])
```

Code 3.1: Store the lines of the geojson

Go through the components and create a raw graph with the trivial edges.

```
1 def trivial_edges(self):
2     for line in self.coordinates:
3         for i in range(len(line) - 1):
4             point1 = tuple(line[i])
5             point2 = tuple(line[i + 1])
6             point1 = self.get_node(point1)
7             point2 = self.get_node(point2)
8             dist = p2p_distance(point1, point2)
9             self.add_edge(point1, point2, weight=dist)
```

Code 3.2: Get Trivial edges

The get node function is a helper function to minimize the number of the nodes by mapping them into one node if the are too close.

```

1 def trivial_edges(self):
2     for line in self.coordinates:
3         for i in range(len(line) - 1):
4             point1 = tuple(line[i])
5             point2 = tuple(line[i + 1])
6             point1 = self.get_node(point1)
7             point2 = self.get_node(point2)
8             dist = p2p_distance(point1, point2)
9             self.add_edge(point1, point2, weight=dist)

```

Code 3.3: Get nodes

NOTE: The GRID SIZE constant is determined manually after multiple occasions of debugging, and it stores the value $= 1 / (3 * \text{DegreeToMeters}) / 1.5 = 2/9$ meters DegreeToMeters is a constant equal to 111148.16049794375 meters which is the equivalent of one latitude / longitude degree. The GRID SIZE value is the base of all computations in this program. The graph is created by checking each line of the trivial edges for intersections. If two lines intersect, they are segmented and interconnected, ensuring that no unnecessary edges are added. In the very last, the doors' labels will get added to the nodes of the graph based on the coordinates.

```

1 def create_graph(self):
2     # Set up the trivial edges
3     self.trivial_edges()
4     # Checks the intersection of each line with all the other
5     # lines
6     for i in range(len(self.coordinates)):
7         line_string1 = self.coordinates[i]
8         for j in range(i + 1, len(self.coordinates)):
9             line_string2 = self.coordinates[j]
10            intersection_point = shapely.intersection(
11                shapely.LineString(line_string1),
12                shapely.LineString(line_string2),
13                grid_size=GRID_SIZE,
14            )
15            if intersection_point.geom_type == "Point":
16                intersection_coords = self.get_node(

```



```

16         (intersection_point.x, intersection_point.y
17         )
18     )
19     for i in range(len(line_string1) - 1):
20         point1 = tuple(line_string1[i])
21         point2 = tuple(line_string1[i + 1])
22         if not shapely.intersection(
23             shapely.LineString([point1, point2]),
24             shapely.LineString(line_string2),
25             grid_size=GRID_SIZE,
26         ):
27             continue
28         point1 = self.get_node(point1)
29         point2 = self.get_node(point2)
30         if self.has_edge(point1, point2):
31             self.remove_edge(point1, point2)
32         if not self.has_edge(point1,
33             intersection_coords):
34             dist = p2p_distance(point1,
35                 intersection_coords)
36             self.add_edge(point1,
37                 intersection_coords, weight=dist)
38         if not self.has_edge(point2,
39             intersection_coords):
40             dist = p2p_distance(point2,
41                 intersection_coords)
42             self.add_edge(point2,
43                 intersection_coords, weight=dist)
44         line_string1.insert(i + 1,
45             intersection_coords)
46         break
47     for i in range(len(line_string2) - 1):
48         point1 = tuple(line_string2[i])
49         point2 = tuple(line_string2[i + 1])
50         if not shapely.intersection(
51             shapely.LineString([point1, point2]),
52             shapely.LineString(line_string1),
53             grid_size=GRID_SIZE,
54         ):
55             continue
56         point1 = self.get_node(point1)

```

```
49         point2 = self.get_node(point2)
50         if self.has_edge(point1, point2):
51             self.remove_edge(point1, point2)
52         if not self.has_edge(point1,
53                               intersection_coords):
54             dist = p2p_distance(point1,
55                                 intersection_coords)
56             self.add_edge(point1,
57                             intersection_coords, weight=dist)
58         if not self.has_edge(point2,
59                               intersection_coords):
60             dist = p2p_distance(point2,
61                                 intersection_coords)
62             self.add_edge(point2,
63                             intersection_coords, weight=dist)
64         line_string2.insert(i + 1,
65                             intersection_coords)
66         break
67
68     # Removing the selflooped edges
69     self.remove_edges_from(nx.selfloop_edges(self))
70
71     # adding doors id to the nodes
72     self.check_door_id()
```

Code 3.4: Create graph function

```
1 def check_door_id(self):
2     for door in self.doors["features"]:
3         for node in self.nodes:
4             if p2p_distance(door["geometry"]["coordinates"],
5                             node) < GRID_SIZE:
6                 self.nodes[tuple(node)]["properties"] = door["
7                     properties"]
8                 break
```

Code 3.5: check door id function

To ensure this process only occurs the first time the app is launched, the following function is implemented to store the generated graph as a JSON file, which can be loaded as needed.

```
1 def graph_to_json(self):
2     data = json_graph.node_link_data(self)
3     json_data = json.dumps(data, indent=4)
4     with open(f"resources/graph_routes{self.floor_nbr}.json", "
5             w") as f:
6         f.write(json_data)
```

Code 3.6: graph to json function

Generation of the main graph:

The use of the application in a selected mode will impact the creation of the general graph, which can be implemented this way for the first 3 graphs. Note: This method removes duplicated nodes in the composed graph to keep the uniqueness of the nodes, not to cause path finding errors.

```
1 def connect_graphs(self, mode):
2     # join graph_00 and graph_0
3     graph_00_to_0 = nx.compose(self.routes[0].graph, self.
4                               routes[1].graph)
5     # join the previous graph with graph_1
6     general_graph = nx.compose(graph_00_to_0, self.routes[2].
7                               graph)
8     general_graph.remove_nodes_from(self.check_disjunction())
9     # extract elevators and stairs
10    elevators, stairs = self.get_connection_points()
11    # relates the elevators of the 3 graphs
12    if mode is not Modes.SOS:
13        for i in range(len(elevators) - 1):
14            for j in range(i + 1, len(elevators)):
15                # one meter radius for overlapping detection
16                if p2p_distance(elevators[i], elevators[j]) <
17                    GRID_SIZE * 9 / 2 * 3:
18                    general_graph.add_edge(elevators[i],
19                                          elevators[j], weight=0)
19    # relates the stairs of the 3 graphs
20    if mode is not Modes.SPECIAL:
21        for i in range(len(stairs) - 1):
22            for j in range(i + 1, len(stairs)):
```

```
20         # one meter radius for overlapping detection
21         if p2p_distance(stairs[i], stairs[j]) <
            GRID_SIZE * 9 / 2 * 3:
22             general_graph.add_edge(stairs[i], stairs[j]
                ], weight=0)
23     self.general_graph = general_graph
24     return self.general_graph
```

Code 3.7: Connecting sub graphs

Path Finding

The NetworkX library offers pre-implemented shortest path finding algorithms; in this case, Dijkstra's algorithm [21] was used as follows: Upon the user's movement, their current position is updated by finding the closest node, on the graph of the current floor, to that position. However, the destination position remains fixed and unchanged.

```
1 def get_shortest_path(self, current_pos, general_graph):
2     current_graph = self.routes[self.student.get_floor + 1].
        graph
3     graph_source = self.get_current_pos(current_graph,
        current_pos)
4     self.get_destination_coord(general_graph, graph_source)
5     graph_dest = self.get_destination
6     self.short_path = nx.dijkstra_path(
7         general_graph, graph_source, graph_dest, weight="weight
            "
8     )
9     self.set_path_to_draw()
```

Code 3.8: Shortest Path Finding

Closest node of a specific destination:

The desired destination may have multiple entrances in the building. To address this, the following function is implemented to identify the closest door among all the destination's entrances.

```
1 @staticmethod
2     def find_closest_dest(list_of_nodes, graph, source):
3         if len(list_of_nodes) > 1:
```

```
4         shortest_paths = {}
5         for coord in list_of_nodes:
6             length = nx.dijkstra_path_length(graph, source,
7                                             coord)
8             shortest_paths[coord] = length
9         return min(shortest_paths, key=shortest_paths.get)
10    else:
11        return list_of_nodes[0]
```

Code 3.9: Closest Destination's door

3.2.3 2D Implementation

To implement the 2D mini-map, the coordinates are extracted from the GeoJSON file and stored for each floor with initial latitude - longitude values. These values will serve, after being scaled, the rendering part of the application.

User Movement:

The user can move freely within the walkable areas, provided their next move doesn't lead them through a wall. The following algorithm addresses this issue.

Point in polygon

[22]

Algorithm 1 Point in Polygon Check

```

1: function POINTINPOLYGON( $x, y, polygon$ )
2:   Input:  $x, y$ : Coordinates of the point to check
3:    $poly$ gon: List of vertices of the polygon
4:   Output: True if the point is inside the polygon, False otherwise
5:
6:    $n \leftarrow$  length of  $poly$ gon
7:    $inside \leftarrow$  False
8:    $p1x, p1y \leftarrow polygon[0]$ 
9:
10:  for  $i = 1$  to  $n + 1$  do
11:     $p2x, p2y \leftarrow polygon[i \bmod n]$ 
12:    if  $y > \min(p1y, p2y)$  and  $y \leq \max(p1y, p2y)$  then
13:      if  $x \leq \max(p1x, p2x)$  then
14:        if  $p1y \neq p2y$  then
15:           $xinters \leftarrow (y - p1y) \cdot (p2x - p1x) / (p2y - p1y) + p1x$ 
16:        end if
17:        if  $p1x = p2x$  or  $x \leq xinters$  then
18:           $inside \leftarrow \neg inside$ 
19:        end if
20:      end if
21:    end if
22:     $p1x, p1y \leftarrow p2x, p2y$ 
23:  end for
24:
25:  return  $inside$ 
26: end function

```

The previous algorithm played a crucial role in implementing the user movement functionality.

```

1  def move_student(self, joystick):

```

```
2     sin_a = math.sin(self.angle)
3     cos_a = math.cos(self.angle)
4     dx, dy = 0, 0
5     speed = STUDENT_SPEED
6     speed_sin = speed * sin_a
7     speed_cos = speed * cos_a
8     direction = self.check_direction(joystick)
9     if direction == "up":
10         dx += speed_cos
11         dy += speed_sin
12     if direction == "down":
13         dx += -speed_cos
14         dy += -speed_sin
15     if direction == "right":
16         dx += speed_sin
17         dy += -speed_cos
18     if direction == "left":
19         dx += -speed_sin
20         dy += speed_cos
21     self.check_collision(dx, dy, self.maps[self.floor + 1].
        world_map)
22
23 def check_collision(self, dx, dy, polygons):
24     can_move = True
25     for polygon in polygons:
26         if point_in_polygon(self.x + dx, self.y + dy, polygon):
27             can_move = False
28             break
29     if can_move:
30         self.x += dx
31         self.y += dy
```

Code 3.10: User movement functions

3.2.4 3D Implementation

The following techniques were used to transition from 2D to 3D:

Ray casting

Algorithm 2 Ray Casting

Function `cast(self, line)`

```

1: Input: line = (x1, y1, x2, y2), self = Ray to cast
2: Output: Intersection point (x, y) or None
3:
4: x1, y1  $\leftarrow$  line[0], line[1]
5: x2, y2  $\leftarrow$  line[2], line[3]
6: x3, y3  $\leftarrow$  pos[0], pos[1]
7: x4, y4  $\leftarrow$  end[0], end[1]
8:
9: // Calculate denominator
10: den  $\leftarrow$  (x1 - x2)  $\cdot$  (y3 - y4) - (y1 - y2)  $\cdot$  (x3 - x4)
11:
12: // Check if lines are parallel
13: if den == 0 then
14:     return None ▷ Lines are parallel or coincident
15: end if
16:
17: // Calculate parameters
18: t  $\leftarrow$  ((x1 - x3)  $\cdot$  (y3 - y4) - (y1 - y3)  $\cdot$  (x3 - x4))/den
19: u  $\leftarrow$  -((x1 - x2)  $\cdot$  (y1 - y3) - (y1 - y2)  $\cdot$  (x1 - x3))/den
20:
21: // Check intersection conditions
22: if 0 < t < 1 and u > 0 then
23:     x  $\leftarrow$  x1 + t  $\cdot$  (x2 - x1)
24:     y  $\leftarrow$  y1 + t  $\cdot$  (y2 - y1)
25:     return (x, y) ▷ Intersection point found
26: end if
27:
28: return None ▷ No intersection found

```

Height mapping

The height of the rendered 3D elements is determined by the distance calculated by the ray casting algorithm. Additionally, the height mapping is achieved through a series of scaling steps: from world coordinates to user coordinates, and finally to wall coordinates.


```
1 def get_wall(self, borders, doors):
2     wall_dist = []
3     points = []
4
5     for room in borders.world_map + borders.outer_line:
6         for i in range(len(room) - 1):
7             p = self.cast(room[i] + room[i + 1])
8             if p is not None:
9                 wall_dist.append(self.ray_distance(p, borders.
10                     scale))
11                 points.append(p)
12
13     if wall_dist:
14         minimal_dist, closest_point = min(zip(wall_dist, points))
15         closest_door = Ray.get_door(doors, closest_point,
16             borders.scale)
17         if closest_door[0] < GRID_SIZE * 1E3:
18             wall_type = closest_door[1]
19         else:
20             wall_type = WALL
21
22     y = (
23         USER_HEIGHT
24         * minimal_dist
25         * math.cos(self.angle_rel)
26         * math.tan(V_FOV / 2 * math.pi / 180)
27     ) # user coords
28     y *= METERS_TO_SCREEN / WALL_HEIGHT # wall coords
29     return y, wall_type
30
31 return None
```

Code 3.11: Get Wall

Note: The previous function also identifies the type of wall the user is currently looking at, such as a door, elevator/stairs, or exit. Consequently, the following functions are defined:

```
1 @staticmethod
2 def door_distance(door, point, scale):
3     scaled_door = (door[0]*scale[0], door[1]*scale[1])
4     scaled_point = (point[0]*scale[0], point[1]*scale[1])
5     return p2p_distance(scaled_point, scaled_door)
6 @staticmethod
```

```
7     def get_door(doors,point,scale):
8         distance = float('inf')
9         door_type = None
10        for v in doors.classrooms.values():
11            for node in v:
12                if Ray.door_distance(node,point,scale) <= distance:
13                    distance = Ray.door_distance(node,point,scale)
14                    door_type = DOOR
15        for node in doors.stairs:
16            if Ray.door_distance(node, point, scale) <= distance:
17                distance = Ray.door_distance(node, point, scale)
18                door_type = STAIR
19        for node in doors.elevators:
20            if Ray.door_distance(node, point, scale) <= distance:
21                distance = Ray.door_distance(node, point, scale)
22                door_type = ELEVATOR
23        for node in doors.exits:
24            if Ray.door_distance(node, point, scale) <= distance:
25                distance = Ray.door_distance(node, point, scale)
26                door_type = EXIT
27        for node in doors.library:
28            if Ray.door_distance(node, point, scale) <= distance:
29                distance = Ray.door_distance(node, point, scale)
30                door_type = DOOR
31        for wc in doors.wc.values():
32            for node in wc:
33                if Ray.door_distance(node, point, scale) <=
34                    distance:
35                    distance = Ray.door_distance(node, point, scale
36                        )
37                    door_type = DOOR
38
39        return distance,door_type
```

Code 3.12: Get Door type

3.2.5 Rendering

Rendering in this application utilizes basic widgets and simple shapes to display scenes, along with the use of colors to texture the 3D elements.

```
1 def render_3D(self, screen):
2     with screen.canvas:
3         Color(0.5, 0.5, 0.5)
4         Rectangle(pos=(0, 0), size=(WIDTH, HALF_HEIGHT))
5
6
7     for ray in screen.student.vision:
8         y, wall_type = ray.get_wall(screen.maps[screen.
9             student.get_floor + 1], screen.doors[screen.
10                student.get_floor+1])
11         if y:
12             Color(
13                 *(
14                     (1, 1, 1) if screen.student.vision.index(ray)
15                     % 2 == 0
16                     else (0.96, 0.7, 0.62)
17                 )
18             )
19             Rectangle(
20                 pos=(screen.student.vision.index(ray) *
21                     RECT_WIDTH, y),
22                 size=(RECT_WIDTH, HEIGHT - 2 * y if HEIGHT
23                     - 2 * y > 0 else 0))
24
25         if wall_type != WALL:
26             Color(
27                 *(
28                     (1, 1, 0) if wall_type == ELEVATOR else
29                     (1,1,0) if wall_type == STAIR else
30                     (0.6, 0.4, 0.2) if wall_type == DOOR
31                     else
32                     (1,0,0)
33                 )
34             )
```

```

32         Rectangle(
33             pos=(screen.student.vision.index(ray) *
34                 RECT_WIDTH, y),
35             size=(RECT_WIDTH, HEIGHT // 2 - y if
36                 HEIGHT // 2 - y > 0 else 0)
37         )
38
39     self.render_floor_label(screen)

```

Code 3.13: 3D Rendering

Modes-Destination Selection Screens Rendering:

For these screens that we can call static, the Renderer class loads up the widgets to the screen, where these widgets have callback function upon interaction.

Mini Map Rendering:

For rendering the Mini map, the Renderer scales real values such as the user position, building coordinates of the current floor, and the path to be drawn, to create a blueprint of the building and display the current status of maps and the user.

```

1  @staticmethod
2      def render_minimap(path, screen):
3          longs, lats = screen.maps[screen.student.get_floor + 1].
4              mini_map_vertices
5          scale_x, scale_y = screen.maps[screen.student.get_floor +
6              1].scale
7          # drawing based on minimap_scale : fitting into 100 by 100
8          scale_x *= MINIMAP_SIZE
9          scale_y *= MINIMAP_SIZE
10         with screen.canvas:
11             # MINI_MAP_ROOMS
12             Color(0, 0, 0)
13             for room in (
14                 screen.maps[screen.student.get_floor + 1].world_map
15                 + screen.maps[screen.student.get_floor + 1].
16                     outer_line
17             ):
18                 pixel_coords = []
19                 for lon, lat in room:
20                     x = (lon - longs[0]) * scale_x
21                     y = (lat - lats[0]) * scale_y
22                     pixel_coords.append((x, y))

```

```
20         Line(points=pixel_coords)
21     # MINI_MAP_ROUTES
22     for i in range(len(path) - 1):
23         Color(0, 1, 0)
24         # path [(x,y),(x,y)...]
25         posx, posy = (path[i][0] - longs[0]) * scale_x, (
26             path[i][1] - lats[0]
27         ) * scale_y
28         endx, endy = (path[i + 1][0] - longs[0]) * scale_x, (
29             path[i + 1][1] - lats[0]
30         ) * scale_y
31         Line(points=(posx, posy, endx, endy), width=1.5)
32     # MINI_MAP_STUDENT
33     Color(1, 0.5, 0.5)
34     for ray in screen.student.vision:
35         # scaling to the minimap
36         posx, posy = (ray.pos[0] - longs[0]) * scale_x, (
37             ray.pos[1] - lats[0]
38         ) * scale_y
39         endx, endy = (ray.end[0] - longs[0]) * scale_x, (
40             ray.end[1] - lats[0]
41         ) * scale_y
42         Line(points=(posx, posy, endx, endy))
```

Code 3.14: Mini map Rendering

Throughout the rendering process, maintaining the application's high performance is achieved by updating screens only upon event dispatching. This approach ensures that screens remain independent of frames per second and only update when user interaction is detected.

3.2.6 Events Handling

In the main loop of the application, the general event listener is called every 60 frames per second to check if any interactive elements on the screen have been interacted with. If an interaction is detected, the conditions of that screen are checked to proceed further.

```
1 def listener(self, dt):
2     if self.screenManager.current == "ModePage":
3         self.mode_page.mode = None
4         if self.mode_page.check_mode():
5             self.screenManager.current = "HomePage"
6             self.renderer.render_welcome_page(
7                 self.home_page, self.mode_page.get_mode
8             )
9         if self.screenManager.current == "HomePage":
10            self.main_page.student.set_pos(*STUDENT_POS)
11            self.main_page.student.update()
12            if self.home_page.form_checked(self.mode_page.get_mode)
13                :
14                self.screenManager.current = "MainPage"
15                self.set_main_page()
```

Code 3.15: General Event Listener

Callbacks

Kivy allows callbacks to be bound to the window for each screen to capture interactions. Below are these callbacks:

```
1     def on_touch_move_update(self, *args):
2         args = list(args)
3         self.main_page.update_joystick_position(args)
4         self.main_page.update_student_vision(args)
5         self.renderer.update(self.main_page)
6
7     def on_touch_down_update(self, dt):
8         if self.main_page.joystick.get_moving():
9             self.main_page.joystick_move()
10            self.renderer.update(self.main_page)
11
12     def on_touch_up_update(self, *args):
```

```
13     args = list(args)
14     self.main_page.joystick.touch_up(args)
15     self.renderer.update(self.main_page)
16
17     def back_button_clicked(self, window, key, *largs):
18         if key == 27:
19             if self.screenManager.current == "HomePage":
20                 self.home_page.screen_clear()
21                 self.screenManager.current = "ModePage"
22                 return True
23             if self.screenManager.current == "MainPage":
24                 self.home_page.clear_homepage_fields()
25                 Window.unbind(
26                     on_touch_move=self.on_touch_move_update,
27                     on_touch_up=self.on_touch_up_update
28                 )
29                 self.main_page.clear_screen()
30                 self.screenManager.current = "HomePage"
31                 return True
```

Code 3.16: Callbacks

3.2.7 Shortcomings

SmartNav Application might be susceptible to implementation shortcomings in the following parts :

- **Rendering Efficiency:** Challenges in maintaining a smooth frame rate while rendering 3D scenes in high definition, like increasing the numbers of emitted rays or handling large datasets.
- **Algorithmic Complexity:** Difficulties in optimizing algorithms, especially when dealing with bad datasets that have to be treated in a certain way in order to provide the needed information.
- **Cross-platform Compatibility:** Issues arising from differences in how the application performs on different platforms or devices. Since the application is designed to work in portrait mode, differences of density of pixels per inch might cause elements to appear in a stretched way.

The desktop app is mainly used for debugging and testing features before building the APK file.

3.3 Testing

3.3.1 Performance Testing

To evaluate the execution efficiency of the algorithms, the Python `timeit` [23] library was used to measure the time in seconds.

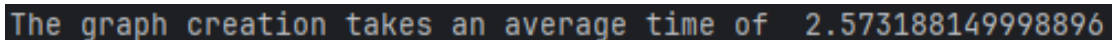
These tests are conducted to assess potential delays caused by certain functions, which could affect the smoothness of the application and its FPS (Frames Per Second) count. The following classes were tested:

Graph Generator

The following test function measures the average time required to create a graph for a specified floor. This is done by calling the `create_graph` function 10 times for a given floor and computing the average execution time.

```
1 def graph_creation_speed(floor_nbr):
2     routes_path = os.path.join(f"../resources/routes{str(floor_nbr)}
3         }.geojson")
4     doors_path = os.path.join(f"../resources/doors{str(floor_nbr)}.
5         geojson")
6     graph = Graph_Generator(routes_path, doors_path, floor_nbr)
7     iteration_nbr = 10
8     print("The graph creation takes an average time of ",
9         timeit.timeit(graph.create_graph,
10             number=iteration_nbr) / iteration_nbr
11     )
```

Code 3.17: Graph testing function



```
The graph creation takes an average time of 2.573188149998896
|
```

Figure 3.17: Graph testing result

Ray

To render the 3D scene, rays emitted from the user's position must undergo a casting function to check for intersections with surrounding polygons. Additionally, the rays cast all the lines that form the floor map. The following test function, checks the average time of creating a **ray casting** function of a Ray.

```
1 def test_cast():
2     ray = Ray((1, 1), math.pi / 180 * 10, 0)
3     iteration_nbr = 10000
4     print("The execution time of 1000 calls of the ray cast
5           algorithm is:", timeit.timeit(
6               lambda: ray.cast(tuple(random.randint(0, 5) for _ in range
7                                   (4))),
8               number=iteration_nbr,
9     ))
```

Code 3.18: ray casting testing function

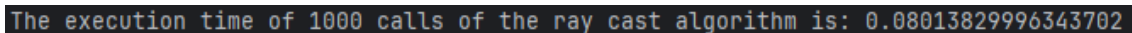


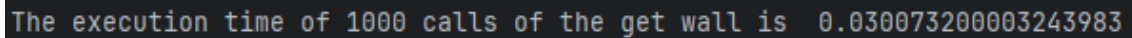
Figure 3.18: Ray-Ray cast testing result

The following test function, checks the average time of **get wall** function of a Ray, where the ray detects the type of point it got from the ray casting. A mock class has been created with mock data in order to test:

```
1 class Borders:
2     def __init__(self, world_map, outer_line, scale):
3         self.world_map = world_map
4         self.outer_line = outer_line
5         self.scale = scale
6 class MockRay:
7     def __init__(self):
8         self.angle_rel = 0
9
10    def ray_distance(self, point, scale):
11        return math.sqrt(point[0]**2 + point[1]**2) / scale
12    @staticmethod
13    def get_door(doors, closest_point, scale):
14        if random.choice([True, False]):
15            return (random.uniform(0, GRID_SIZE * 1E3), DOOR)
```

```
16         else:
17             return (GRID_SIZE * 1E3 + 1, WALL)
18
19 borders = Borders(
20     world_map=[
21         [(0, 0), (0, 5), (5, 5), (5, 0)],
22         [(10, 0), (10, 5), (15, 5), (15, 0)]
23     ],
24     outer_line=[(0, 0), (0, 10), (10, 10), (10, 0)],
25     scale=(1.0,1.0)
26 )
27 doors = [(3, 3), (12, 3)]
28 ray = MockRay()
29 print("The execution time of 1000 calls of the get wall is ",timeit
    .timeit(
30         lambda: ray.get_wall(borders, doors),
31         number=1000,
32     ))
```

Code 3.19: Get wall testing function



```
The execution time of 1000 calls of the get wall is 0.030073200003243983
```

Figure 3.19: Ray-Get Wall testing result

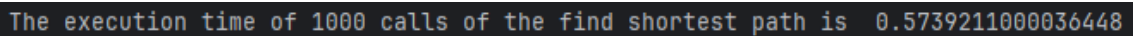
Path

While using the app, the user moves freely on the map, prompting the path class to search for a new shortest path with each movement. This necessitates testing the time required for the class to generate the new path. To evaluate the Path class and its shortest path algorithm, a mock Path class was created. The shortest path function was tested by running it 1000 times with various source and destination nodes on the graph:

```
1 class MockPath:
2     def __init__(self, graph):
3         self.graph = graph
4     def find_shortest_path(self):
5         current_pos, destination_pos = random.choice(list(self.
            graph.nodes)), random.choice(list(self.graph.nodes))
6         self.short_path = nx.dijkstra_path(
```

```
7         self.graph, current_pos, destination_pos, weight="
           weight")
8 def get_graph():
9     with open(f"../resources/graph_routes1.json", "r") as f:
10         data = json.load(f)
11         return nx.node_link_graph(data)
12 graph = get_graph()
13 path = MockPath(graph)
14 def test_path_finding():
15     print("The execution time of 1000 calls of the find shortest
           path is ", timeit.timeit(
16         path.find_shortest_path,
17         number=1000,
18     ))
19 test_path_finding()
```

Code 3.20: Path testing function



```
The execution time of 1000 calls of the find shortest path is 0.5739211000036448
```

Figure 3.20: Path-Shortest Path testing result

Student

Each movement action initiated by the user undergoes a checking process to ensure that the direction of movement does not allow the user to pass through a restricted area.

To evaluate the time efficiency of the collision detection function in the Student class, the function was tested by running it with a set of polygons 1000 times.

```
1 class MockStudent:
2     def __init__(self, polygons):
3         self.x = 1
4         self.y = 1
5         self.polygons = polygons
6
7     def check_collision(self):
8         dx, dy = (random.randint(10, 20), random.randint(10, 20))
9         can_move = True
10         for polygon in self.polygons:
11             if point_in_polygon(self.x + dx, self.y + dy, polygon):
```

```
12         can_move = False
13         break
14     if can_move:
15         self.x += dx
16         self.y += dy
17
18 polygons = [
19     [(10, 10), (12, 10), (11, 12)],
20     [(5, 5), (5, 6), (6, 6), (6, 5)],
21     [(20, 20), (21, 21), (22, 20), (21, 19), (20, 19)],
22     [(15, 15), (16, 16), (17, 15), (17, 14), (16, 13), (15,
23         14)],
24     [(30, 30), (32, 31), (31, 32), (29, 31), (30, 29)]
25 ]
26 user = MockStudent(polygons)
27 def test_collision_detection():
28     print("The execution time of 1000 calls of the collision
29         detection is is ", timeit.timeit(
30         user.check_collision,
31         number=1000,
32     ))
33 test_collision_detection()
```

Code 3.21: Collision detection testing function

```
The execution time of 1000 calls of the collision detection is 0.019498799927532673
```

Figure 3.21: Collision detection testing result

3.3.2 Manual Testing

GUI Testing

To evaluate the application's behavior under continuous random input, SmartNav undergoes Manual GUI testing. To test user movement, directions are randomized while monitoring screen updates for the 3D rendering and the 2D mini-map, multiple clicks on the screen were performed. The resulting scenarios of the testing come as follows:

- **Accurate Rendering and Synchronization :**

- **3D Rendering :** The Rendering accurately reflects the user's movement and surroundings.

The user observes the walls of the building when their position is sufficiently close to make the walls appear visible. However, the walls disappear when the distance exceeds a certain limit to prevent artifacts caused by negative pixels.

- **2D Mini Map :** The mini map is updated correctly upon movement and the path is continuously providing the right shortest path. When there is a difference in floors between the current position and the destination, the path is first drawn from the current position to the connecting point. After changing floors, a new path is then drawn from the user's position to the destination.

Additionally, transitioning between floors updates the mini-map to reflect the layout of the new floor.

- **Responsive User Movement :** The application responds to random directions on the screen without delay.

Left-right swipes within the designated zone rotate the user's vision in the correct direction.

Joystick movements initiate user movement in the correct direction. However, movement through walls is restricted by collision detection, and the user can only pass through exits.

- **No Glitches or Visual Artifacts :** The application runs without any visual artifacts or anomalies.

Manual Path Testing

Path - mode compatibility

To validate the path generation feature, paths are generated based on the selected mode to ensure compatibility without conflicting with the preferred use case. Specifically, connection points between floors are restricted based on the chosen mode: elevators for accessible mode and stairs for emergency mode.

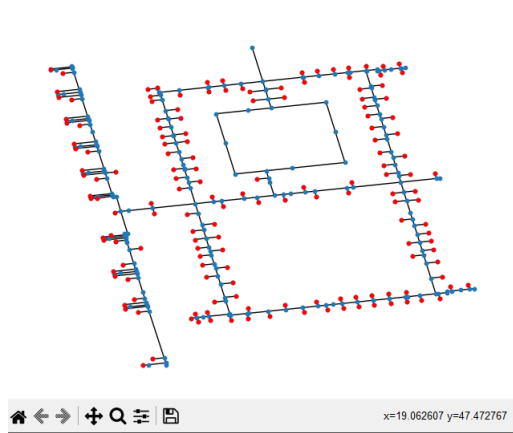
Testing this functionality involved selecting a mode and simulating scenarios to trigger path generation. The application successfully responded by providing paths that include the correct connection points based on the selected mode.

Shortest path finding efficiency

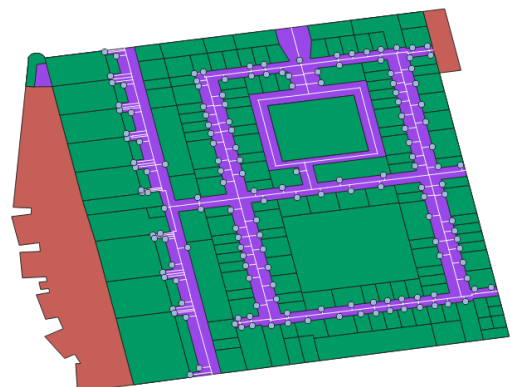
An additional test is conducted on the shortest path algorithm by randomly selecting classrooms or building facilities each time, ensuring that the Path class consistently provides the shortest path to the desired destination without errors.

Graph Creation Testing

To ensure the graphs were created accurately and without errors, we used the Matplotlib [24] library for plotting and the NetworkX library for drawing the graphs. To verify that the doors of the building were correctly assigned to the nodes, we compared the drawn graphs with maps visualized in QGIS software. In this case, the plotting will color the door nodes with red color, otherwise blue is used. This manual verification process ensured that the app correctly handled the geoJSON files and accurately extracted the necessary data.



(a) Floor 00 Graph



(b) Floor 00 QGIS Map

Figure 3.22: Floor 00 Testing

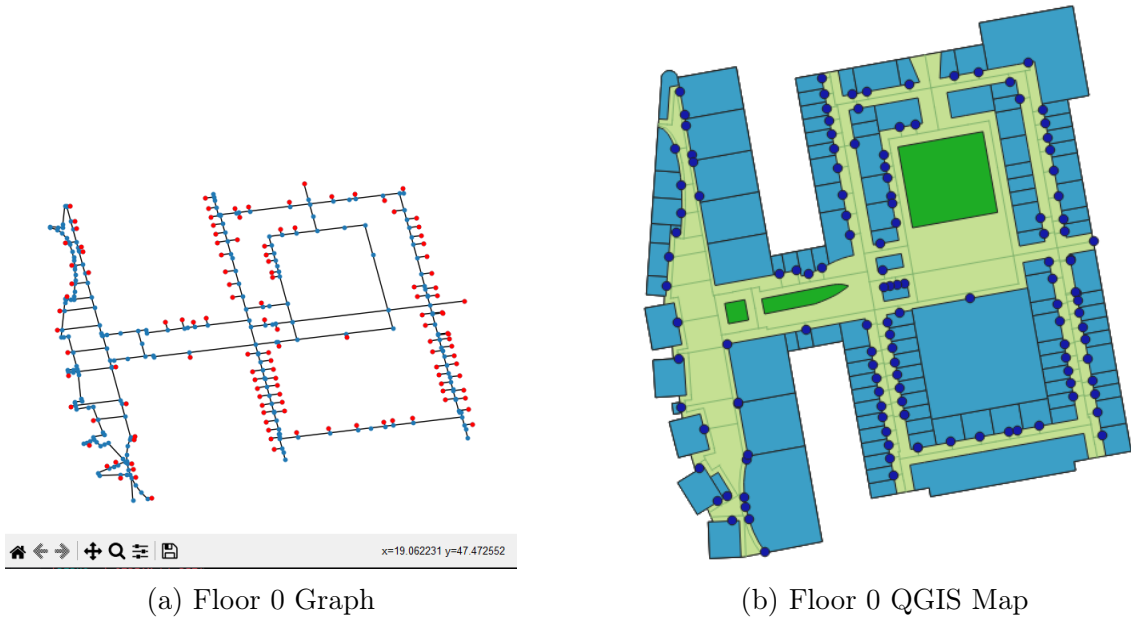


Figure 3.23: Floor 0 Testing

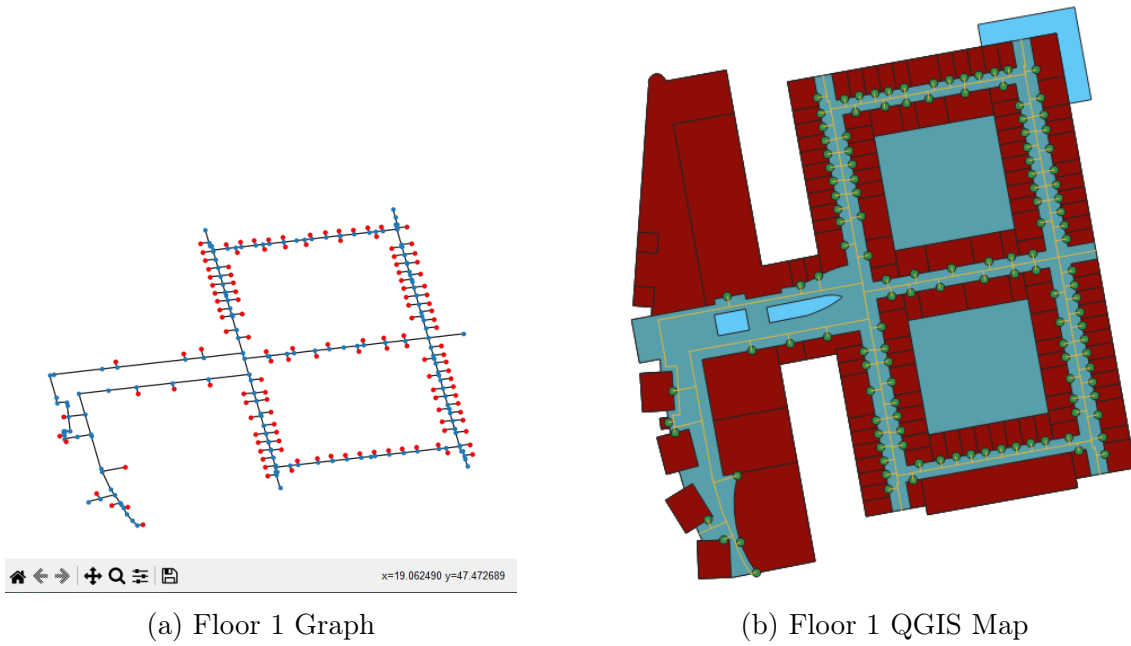


Figure 3.24: Floor 1 Testing

The previous test helped in debugging issues related to the grid size constant, which determines how closely nodes should be situated to merge into a single node. Manual debugging led to the determination of this constant to ensure that routes intersect properly and to prevent unnecessary edges in the graph when multiple nodes are closely positioned, which could otherwise lead to unnecessary loops.

During the application's execution, the doors on the map were visualized as colored circles. This was achieved by mapping each door type to a specific color using a dictionary.



Figure 3.25: Doors drawing Testing

Graph Connectivity Testing

To verify the connectivity of the graph generated depending on the mode of use, a test is performed in the loader after calling the `connect_graph` function. This test checks whether the generated graph is connected or not.

```
This is the Normal Mode. Is the Graph Connected ? : True
Graph with 1030 nodes and 1066 edges
This is the Accessible Mode. Is the Graph Connected ? : True
Graph with 1030 nodes and 1053 edges
This is the Emergency Mode. Is the Graph Connected ? : True
Graph with 1030 nodes and 1062 edges
```

Figure 3.26: Graph Connectivity Results

APK Testing

To test the APK file, the app was executed in BlueStacks software as a simulated Android environment, providing logs for the analysis of error messages and warnings. This method helped in identifying issues related to app execution on a mobile device, facilitating the discovery of needed dependencies—both implicit and transitive—that were not explicitly mentioned in the setup instructions.

Chapter 4

Conclusion

In conclusion, this application was developed to address an existing issue that everyone has noticed with navigation within the Faculty of Informatics building at Eötvös Loránd University. It aims to assist in solving this issue, particularly for urgent situations and newcomers, despite its current limitations in reliability. Many improvements are planned for this project to enhance its functionality and make it publicly available for download, ensuring it is useful for anyone who needs it. Additionally, there are future plans to extend the app's capabilities to connect the south and north buildings, making it beneficial for the entire campus, not just the south building.

Creating a 3D application has allowed me to apply what I've learned during my computer science degree. It deepened my understanding of 2D to 3D conversion techniques and the complexities of mobile development. Exploring this field has been particularly interesting and has expanded my knowledge in ways that will be valuable in my future career.

Bibliography

- [1] Android. *armeabi-v7a*. 2024. URL: <https://developer.android.com/ndk/guides/abis#v7a>.
- [2] Android. *arm64-v8a*. 2024. URL: <https://developer.android.com/ndk/guides/abis#arm64-v8a>.
- [3] Python Software Foundation. *Python Programming Language*. 2024. URL: <https://www.python.org/>.
- [4] Python packaging authority. *Pip*. 2024. URL: <https://pypi.org/project/pip/>.
- [5] Canonical. *Ubuntu OS*. 2024. URL: <https://ubuntu.com/>.
- [6] Debian. *Debian GNU / LINUX*. 2024. URL: <https://www.debian.org/>.
- [7] Buildozer. *Buildozer: Generic Python packager for Android and iOS*. 2024. URL: <https://buildozer.readthedocs.io/en/latest/index.html>.
- [8] Android. *Android SDK*. 2024. URL: <https://developer.android.com/tools#tools-sdk>.
- [9] Android. *Android NDK*. 2024. URL: <https://developer.android.com/ndk>.
- [10] Wikipedia. *Lexicographic Order*. 2023. URL: https://en.wikipedia.org/wiki/Lexicographic_order.
- [11] Wikipedia. *Model–View–Controller*. 2023. URL: <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>.
- [12] QGIS Development Team. *QGIS: A Free and Open Source Geographic Information System*. 2023. URL: <https://www.qgis.org/en/site/>.
- [13] Eötvös Loránd University. *Geoinformatics MSc Program*. URL: <https://terkep.elte.hu/>.

- [14] GeoJSON Project. *GeoJSON Format Specification*. URL: <https://geojson.org/>.
- [15] Wikipedia. *Ray Casting*. 2023. URL: https://en.wikipedia.org/wiki/Ray_casting.
- [16] Wikipedia. *Doom (1993 Video Game)*. 2023. URL: [https://en.wikipedia.org/wiki/Doom_\(1993_video_game\)](https://en.wikipedia.org/wiki/Doom_(1993_video_game)).
- [17] Wikipedia. *Taxicab Geometry*. 2023. URL: https://en.wikipedia.org/wiki/Taxicab_geometry.
- [18] Kivy. *Kivy: Open source Python library for rapid development of applications*. 2024. URL: <https://kivy.org/>.
- [19] NetworkX Developers. *NetworkX Documentation*. 2024. URL: <https://networkx.org/documentation/stable/index.html>.
- [20] Shapely. *Shapely: Manipulation and analysis of geometric objects*. 2024. URL: <https://shapely.readthedocs.io/>.
- [21] Wikipedia. *Dijkstra's Algorithm*. 2023. URL: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.
- [22] Wikipedia. *Point in Polygon*. 2023. URL: https://en.wikipedia.org/wiki/Point_in_polygon.
- [23] Python Software Foundation. *timeit — Measure Execution Time of Small Code Snippets*. 2024. URL: <https://docs.python.org/3/library/timeit.html>.
- [24] the Matplotlib development team. *Matplotlib: A 2D Graphics Environment*. 2007. URL: <https://matplotlib.org/>.

List of Figures

2.1	Application Icon	8
2.2	Loading Screen	9
2.3	Mode selection screen	11
2.4	Destination selection screen Normal / Accessible mode	12
2.5	Floors drop down expanded	13
2.6	Classrooms drop down expanded	14
2.7	Other facilities drop down expanded	14
2.8	Select a floor Error	15
2.9	Select a destination Error	16
2.10	Green background floor	17
2.11	Red background floor	17
2.12	Destination selection screen Emergency mode	18
2.13	Doors Indications	20
2.14	2D Minimap	21
2.15	Path User - Connection Point	22
2.16	Path User - Destination	22
2.17	Joystick	23
2.18	Floors indication.	24
2.19	Connection point popup	24
2.20	Reached destination popup	25
3.1	Packages Diagram	27
3.2	Wireframes	29
3.3	Faculty's floor 1 wall map	31
3.4	Faculty maps processing	32
3.5	QGIS interface	32
3.6	QGIS floor representation	33
3.7	Graphs of floors	34

3.8	Doom Game	35
3.9	Ray casting example	36
3.10	The user's Horizontal FOV	37
3.11	Height and Vertical Field of view	38
3.12	Taxicab geometry	39
3.13	QGIS layer to export	41
3.14	Selecting the Coordinate system	42
3.15	Precision selection	42
3.16	Example Geojson	43
3.17	Graph testing result	61
3.18	Ray-Ray cast testing result	62
3.19	Ray-Get Wall testing result	63
3.20	Path-Shortest Path testing result	64
3.21	Collision detection testing result	65
3.22	Floor 00 Testing	67
3.23	Floor 0 Testing	68
3.24	Floor 1 Testing	68
3.25	Doors drawing Testing	69
3.26	Graph Connectivity Results	69

List of Tables

2.1	Application modes and their explanation	10
2.2	UI Elements and their actions	12

List of Algorithms

1	Point in Polygon Check	51
2	Ray Casting	53

List of Codes

2.1	Tools Command	5
2.2	requirements.txt	6
2.3	Requirements installation	6
2.4	Buildozer Installation Command	6
2.5	Buildozer init	6
2.6	Path export	7
2.7	APK building	7
3.1	Store the lines of the geojson	44
3.2	Get Trivial edges	44
3.3	Get nodes	45
3.4	Create graph function	45
3.5	check door id function	47
3.6	graph to json function	48
3.7	Connecting sub graphs	48
3.8	Shortest Path Finding	49
3.9	Closest Destination's door	49
3.10	User movement functions	51
3.11	Get Wall	54
3.12	Get Door type	54
3.13	3D Rendering	56
3.14	Mini map Rendering	57
3.15	General Event Listener	59
3.16	Callbacks	59
3.17	Graph testing function	61
3.18	ray casting testing function	62
3.19	Get wall testing function	62

3.20 Path testing function	63
3.21 Collision detection testing function	64