Side-Eye

Blaser Christian[a], Eckert Niklas[a], Hunziker Carlos[a], Zemp Estelle[a]

[a]*IKG - Institute of Cartography and Geoinformation*

Abstract

Side-Eye is the semester project of group 8 for the lecture *GTA - Geoinformationstechnologien und -analysen*. The objective was to develop an app which combines opensource and user-generated data in a geospatial analysis. In the project Side-Eye, a webapp was implemented which allows the user to track a trip. After completing the tracking, the app will then show them what interesting places they passed and might have missed while they were on the go.

Contents

1. Introduction

The aim of this project is to apply the knowledge acquired during the lectures and labs of GTA in a practical project in order to get to know the entire workflow of a GIS project. IKG (2023) To this purpose, we implemented an app with GIS functionalities. This report will explain the developped app Side-Eye[1] and illustrate how it works.

1.1. Research Question

At the core of this project lies a trivial, every-day problem: The user is glued to their phone or preoccupied with work and thus blind to their surroundings while travelling. Alternatively, they might be driving and focused on traffic, which makes it impossible to look up interesting things they see. This is especially prevalent while commuting. In order to create a solution for this, an app was implemented which allows the user to track a trip and then look at interesting points or sights they passed and might have missed.

The app Side-Eye collects geodata, both single points and human trajectory, and combines them with publicly available data from OpenStreetMap (OSM). The data is then stored via Geoserver in a Post-GIS database, automatically analyzed and visualized on the webapp.

The research question to be answered in this report is the following:
”How can self-recorded trajectories be combined with open-source data in a geospatial analysis while optimizing the user interface and experience?”

1.2. Related Work

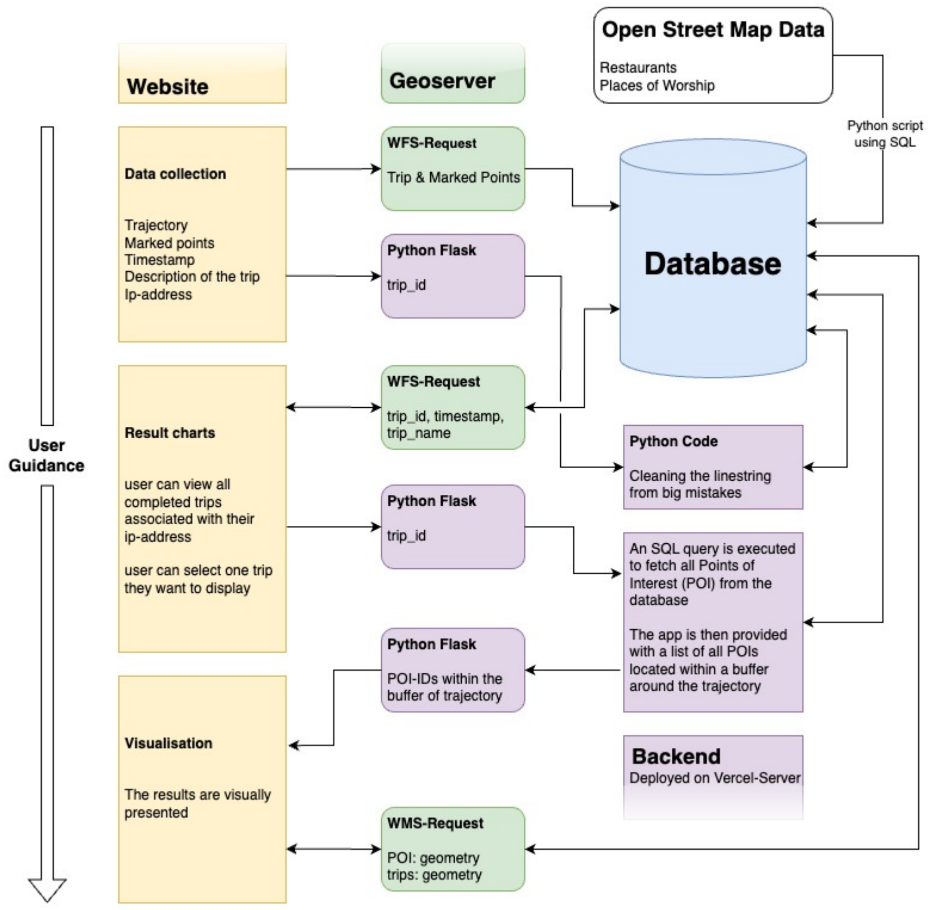
By answering this research question, our work will of course not contribute to filling any existing research gap in the GIS-field. The research question is not novel and has been solved before by many apps similar to ours. Never the less, the project will aid our personal gain of knowledge and experience.

[1]Link to the app: https://n.ethz.ch/ cblase/gta/index.html

2. Data and Methodology

2.1. Overview

An overview of the entire project workflow is provided in Figure 1. There are multiple parts at play in collecting and processing the data.



Open Street Map Data
Restaurants
Places of Worship
Website
Geoserver
Python script using SQL
Data collection
Trajectory
Marked points
Timestamp
Description of the trip
Ip-address
WFS-Request
Trip & Marked Points
Python Flask
trip_id
Database
WFS-Request
trip_id, timestamp, trip_name
User Guidance
Result charts
user can view all completed trips associated with their ip-address
user can select one trip they want to display
Python Code
Cleaning the linestring from big mistakes
Python Flask
trip_id
An SQL query is executed to fetch all Points of Interest (POI) from the database
The app is then provided with a list of all POIs located within a buffer around the trajectory
Python Flask
POI-IDs within the buffer of trajectory
Visualisation
The results are visually presented
Backend
Deployed on Vercel-Server
WMS-Request
POI: geometry
trips: geometry

Figure 1: Project Workflow

On the website, the trajectory, marked points as well as a timestamp and the IP address of the user are collected. When the tracking is stopped, this data is stored in the PostGIS Database via a WFS-Request (WFS - Web Feature Service) and the Geoserver. After the successful upload, the trip ID is sent via Flask to a python function which cleans the trajectory from big jumps in the GPS-tracking and updates the database with the improved version.

On a different page of the website, the user can view all their completed trips. To display this information, the website sends a WFS-Request via Geoserver to the database, which retrieves all trip IDs associated with the user's IP address. Now, the user can select one trip they want to display. The ID of this trip is sent to the python backend, which is deployed on Vercel, for processing of the data.

The python code called by Flask receives the trip ID when the desired entry on the result chart is clicked. Subsequently, an SQL query is executed to fetch all POIs (Points of Interest) in the vicinity of the trip from the database. The app is then provided with a list of all POI IDs located within a buffer around the trajectory.

With the list obtained from Flask, a WFS-Request to the Web Map Service (WMS) is performed on the database, retrieving the geometry of all POIs in this list as well as the trajectory with the current trip ID and the marked points. They are then visualized on the webapp.

2.2. Database Conceptualization

The conceptualization of the database is depicted in the UML-Diagram in Figure 2. The database contains both open-source data from OSM and data that is collected through the app.

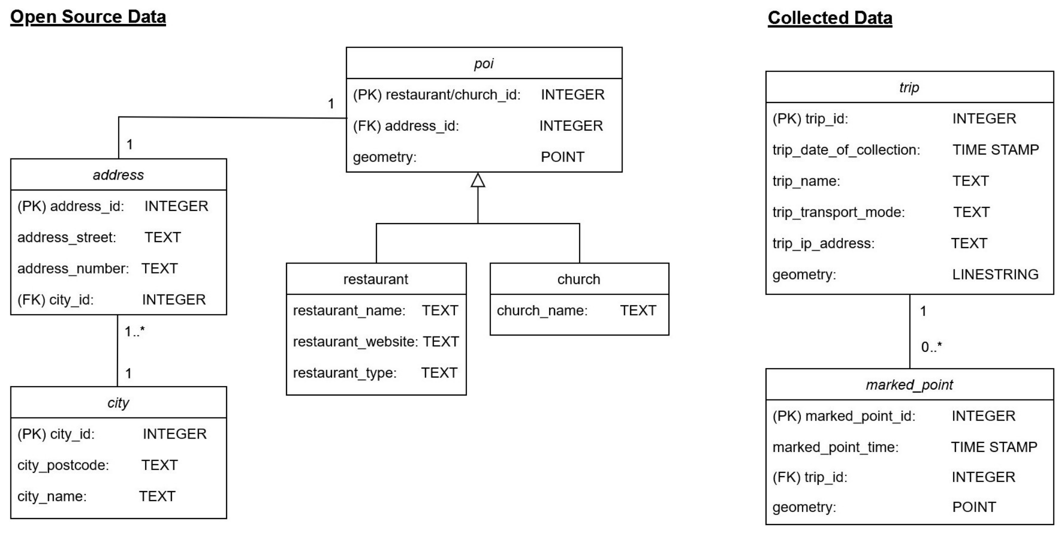


Figure 2: UML-Diagram

There are separate tables for both supported types of POI: one table for restaurants and one for churches (the latter including not only churches but also other places of worship). However, in the UML-Diagram they are visualized as subtypes inheriting from the same class POI, which does not exist as a table in the database. This is to show how other types of POI could be integrated in the app, for example museums or historical buildings.

To achieve database normalization and consequently reduce data redundancy, the addresses as well as the city are stored in separate tables. They are connected to the POI tables through the foreign-keys *address_id* and *city_id*. With this, the database fulfills the third normal form.

On the side of the collected data, there is a table for trips which can be matched to the user by the IP address used to upload them. Marked points are associated to the trip they were recorded in via the foreign-key *trip_id*. Other than this, there are no constraints.

2.3. Open-Source Data

The open-source data was downloaded from OSM via overpass-turbo.eu as a .csv file. In the case of this project there is a table for places of worship and one for restaurants in the city of Zurich. Of course, this could be extended to other categories of POIs as well as a bigger perimeter.

It was possible to choose directly in the query which attributes should be downloaded. The data was then cleaned up using a python function with the pandas library. After that, it was loaded directly into the postGIS-database using the Psycopg2 library in python.

2.4. Movement Recording and Processing

The trajectory is recorded using the *navigator.geolocation.watchPosition()* function in JavaScript. The coordinates (in WGS84) are first stored locally and then uploaded to the database as a GML-Linestring. When this upload was successful, a clean-Linestring function is called on Vercel and given the ID of the uploaded trip.
This function gets the trajectory from the database, corrects big jumps due to GPS-jitters and then reuploads the cleaned version to the database. It works by analysing the distance between two points and deleting points that have a distance above a certain value, which indicates an unrealisticly big jump in the GPS-tracking.

The marked points are uploaded as GML-Points with timestamps of their collection into the Database, as soon as the associated trip is uploaded to the database. The upload of the associated trip returns the primary key ID which is then used to reference the marked points via foreign key.

When the upload is completed, the user has to switch from the map page to the commutes page and find their trip in the list. After clicking on the trip, the user selects the category of their interest. It then sends the trip ID to the backend, which is deployed on Vercel. There, the Linestring is buffered and intersected with all POIs of the selected category to find the points in the vicinity of the route travelled.

The IDs of those points are then returned in a list to the webapp and used to visualize those points through a WFS-request to the database and the Geoserver. As well as the POIs, the trajectory and the marked points are visualized.

All these different processing steps are visualized as a flowchart in Figure 3.

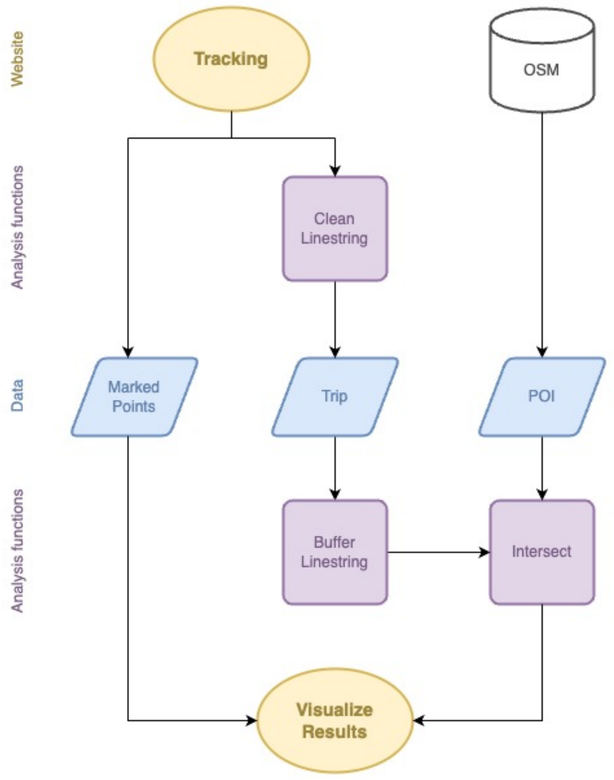


Figure 3: Processing Workflow

The website is split into three main pages:

- Map: display current position, track trips
- Commutes: list completed trips, show results
- About: describe functionality

On the **map page** there are three buttons. Most importantly, there is the start-stop-button to activate and deactivate the tracking mode. When the app is tracking, the text in the button switches from *START* to *STOP* and a blinking red dot appears in the upper right corner of the screen.

Next to this start-stop-button, there are the center-button, which resets the mapview to be centered around your current position, and the pin-button, which allows you to add a marked point. (See Figure 4)

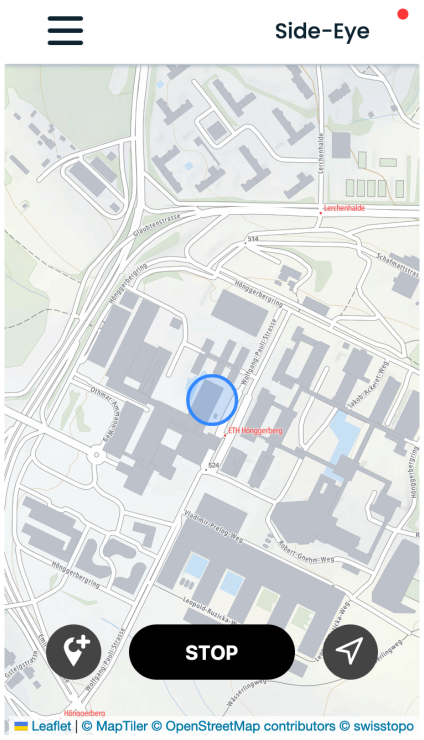
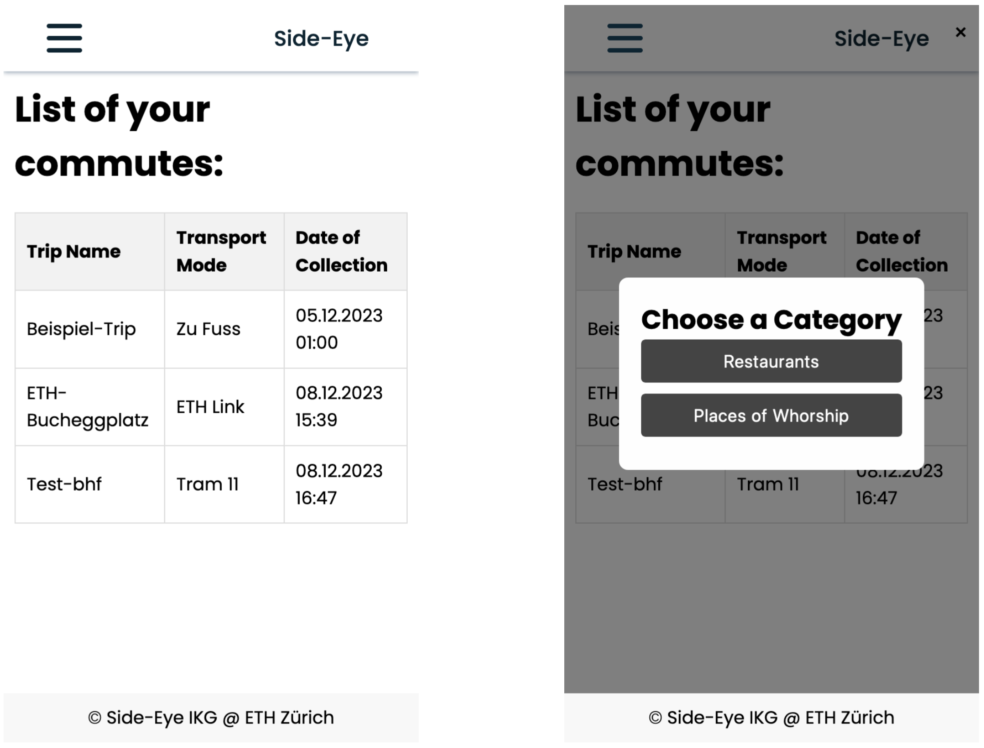

Side-Eye
STOP
Leaflet | © MapTiler © OpenStreetMap contributors © swisstopo
ETH Hönggerberg
Lerchenhalde

Figure 4: Map page in tracking-mode

On the **commutes page** there is a list of all trips recorded with this IP address as well as a few example trips to demonstrate the functionality of the app. Clicking on a trip opens a pop-up, which allows the user to choose a category of POI. (see Figure 5)



List of your commutes:

Trip Name	Transport Mode	Date of Collection
Beispiel-Trip	Zu Fuss	05.12.2023 01:00
ETH-Bucheggplatz	ETH Link	08.12.2023 15:39
Test-bhf	Tram 11	08.12.2023 16:47

Figure 5: Commutes page

After entering the category, the input is processed and displayed on a map. This resulting map is shown and explained in Chapter 3 of this report.

The **about page** of the website explains the basic functionality of Side-Eye and provides contact information.

While designing the app, special attention was payed to **user guidance**. We tried to anticipate unexpected user inputs and interrupt them with pop-up windows instructing the user on how to use the app. For example, when the user tries to mark a point, when they are not currently tracking a trip, there is a pop-up instructing them to first start the recording. Similarly, if a user tries to upload a trip to the database which has only one logged point, they get an error message.

As a basemap we chose the swisstopo LBM from Maptiler. The color scheme for the webapp was developed based on this map. The app is mostly black and white with some shades of grey. Furthermore, some colors were added: a light green as an accent color, blue for positioning and red for making the user aware, that they are tracking (see Figure 6). All these colors also exist in the basemap.

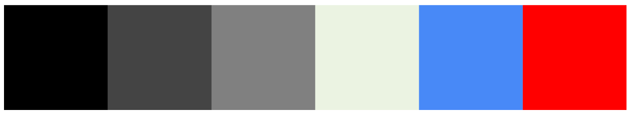


Figure 6: Color scheme as a color ramp

3. Results

The outcome of the analysis is a map visualizing the route of the trip as well as the POIs and marked points. The POIs are visualized as black and the marked points as blue. The map is interactive and displays the names of the POI when the corresponding tags are clicked on (see Figure 7). In the case of restaurants, the pop-up also includes a link to the website if one is available in the database.

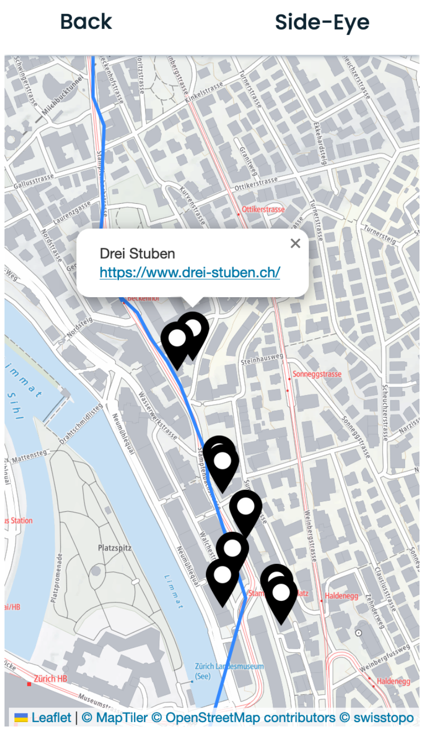

Back
Side-Eye
Drei Stuben
https://www.drei-stuben.ch/
Leaflet | © MapTiler © OpenStreetMap contributors © swisstopo

Figure 7: Visualization of Analysis Results

Because the process to display this information takes some time - especially on the smartphone - a loading icon was added. The icon is shown and rotates as long the JavaScript is still running.

4. Discussion and Conclusion

The proposed research question *"How can self-recorded trajectories be combined with open-source data in a geospatial analysis while optimizing the user interface and experience?"* was explored and answered over the course of the last few weeks. The process of developing the app helped us learn about the handling of geodata as well as the more formal aspects of programming like how to deploy python code on a server.

There are a few aspects in the implementation of Side-Eye which are not yet ideal. For example, the way the user accesses the results of the analysis: Ideally, the result map would pop up right after the trip is uploaded, instead of having to make a detour over the list on the commutes page.
Also, user identification through IP address is not the optimal solution, as the IP address changes. A system with user accounts would perhaps make more sense. Alternatively, it could also be solved with cookies.

Further testing of the app would almost certainly reveal unexpected user inputs and behaviors we did not yet anticipate. With more time, this could be addressed and the user experience improved.

The app in its current state is also quite simple. However, there are many ways it could be extended:

The first being the perimeter. Right now, the app only works in the city of Zurich, because there are only POIs within this perimeter in the database. More open source data could be added easily. Also, more categories of POIs could be added to the analysis.
Another possibility is to add a flexible buffer, which would adjust the buffered distance depending on the population density of the areas travelled. Alternatively, the buffer distance could also be taken as a user input, which would make the result map more dynamic.

In conclusion, there is potential to improve this project. However, we are happy with the results accomplished in the limited timeframe.

CRediT Author Statement

Blaser Christian: Conceptualization, Methodology, Software, Visualization, Writing - Review & Editing.
Eckert Niklas: Supervision, Conceptualization, Methodology, Software, Data Curation, Writing - Review & Editing.
Hunziker Carlos: Conceptualization, Methodology, Software, Project administration, Writing - Review & Editing.
Zemp Estelle: Conceptualization, Methodology, Software, Writing - Original Draft, Writing - Review & Editing.

References

IKG. GTA - Project Description, 2023.