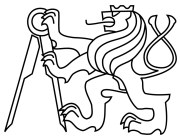
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF BACHELOR'S THESIS

Title:	CMS for the CTU navigator
Student:	Jan Kodera
Supervisor:	Ing. Jiří Chludil
Study Programme:	Informatics
Study Branch:	Information Systems and Management
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2017/18

Instructions

The goal of the thesis is the implementation of a Content Management System (CMS) for the CTU Navigator based on open-source ModX. The primary goal is to provide administration of map schemes, routes, waypoints, and panoramatic views.

1. Analyze
- the conversion of walls, rooms, and suggested waypoints from bitmap formats (jpg, png) to the vector format (svg) and extraction of map schemes,
- filtering useless curves in extracted schemes,
- creating use case spec., domain/class diagram, FURPS.

2. Design
- a data structure and API for mobile clients,
- a GUI for defined user roles - system admin., content admin.,
- an API for mobile clients (clients work offline),
- an adaptor for 3-side services (Google Cal., OpenStreetMaps),
- an administration GUI of buildings (floors, routes, walls, points).

3. Implement
- the API for mobile clients,
- the GUI for map administration,
- the GUI for sending push notifications.

4. Perform appropriate tests - GUI, system performance, etc.

References

Will be provided by the supervisor.

L.S.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague March 7, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Bachelor's thesis

CTU Navigator III - Backend

Jan Kodera

Supervisor: Ing. Jiří Chludil

3rd January 2017

Acknowledgements

I would like to thank my supervisor Jiří Chludil for advice and guidance when writing my thesis and my consultant Michal Maněna for help during its implementation and for mediation of team meetings. I would also like to thank the rest of the team around the CTU Navigator project, Stanislav Mikeš, Jakub Homolka, Peter Janička and Jevheniy Horvát for making this project happen. Finally I would like to thank my family for their support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 3rd January 2017 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Kodera, Jan. *CTU Navigator III - Backend*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Tato bakalářská práce je zaměřena na problematiku návrhu a implementace serverové části informačního systému ČVUT Navigator. Systém je navržen pro podporu navigace uvnitř budov a uzavřených kampusů, s cílem prvotního nasazení v rámci ČVUT pro usnadnění nalezení učeben ČVUT studenty pouze základě kódových označení těchto místností. Součástí práce jsou metody a nástroje pro převedení plánů budov do mapových formátů, tvorba navigačních grafů nad nimi a předání balíčku těchto dat mobilním aplikacím v rámci dokumentovaného API. Systém také podporuje kontrolu přístupových práv k úpravám dat. Celé řešení je integrováno za použití PHP frameworku MODX.

Klíčová slova Indoor navigace, Konverze obrázků do mapových formátů, Navigační graf, Správa přístupových práv, Informační systém

Abstract

This bachelors thesis deals with server side of CTU Navigator information system. The system is designed for navigation inside buildings and closed campuses. It primarily aims at helping CTU students locating classrooms while having only the classroom ID. The thesis consists of methods and tools for converting building floor plans into map formats, using them for building navigation graphs and transferring these data packages to mobile applications via a documented API, as well as integrating data administration and user and access control. The whole solution builds on the MODX PHP framework.

Keywords Indoor navigation, Conversion of images into map formats, Navigation graph, Access rights management, Information system

Contents

List of Figures

Introduction

Every one of us has once found themselves in unfamiliar surroundings, trying to find a place they have never been to before. This task was traditionally solved using paper maps and other "mechanical" tools but with the rise of electronic devices and software solutions another set of helping gadgets, called navigators and later smartphones, emerged. These greatly eased the mentioned navigation problem, yet almost all of them deal only with outdoor spaces, urban streets and country areas. Very few tools help with navigation in larger indoor complexes of closed buildings and/or whole campuses, such as hospitals, schools, government or corporate. This already not too simple effort is made even harder by the fact that most localization technologies, on which all but a few electronic devices for outdoor navigation are built on, such as GPS, do not work in closed spaces, corridors and rooms. New CTU students often reach the conundrum of being provided a room code of a classroom in which their next lesson will take place and have to reach it in relatively short amount of time. Even with older students' assistance it is often difficult to find the target room due to the sheer amount of buildings, floors, and rooms split in several locations which are often designated just by their building and room numbers. This complexity makes them nearly impossible to navigate without a lot of guesswork, luck or inside knowledge. Solving this problem was a goal of team assembled around the CTU Navigator project, an information system designed to provide path tracing and navigation in both indoor and outdoor locations of the school campus. Within this thesis I had worked on the project's backend, which handles conversion of floor plans into map file formats, supports creation of navigation graph on top of them and provides several types of other helper metadata to the mobile apps. Other members of the team were Stanislav Mikeš performing project management and building Windows 10 Mobile app, Jakub Homolka analyzing the problem domain, Peter Janička working on pathing algorithms and Android mobile app and Jevhenij Horvat handling the corresponding web client.

Chapter 1

State-of-the-art

1.1 Indoor navigation

Navigation systems, as we know them today, started in military setting [1]. In 1973, during the bloom of the Cold War between the USA and Soviet Union, US military needed a new way to navigate military units and technologies across foreign spaces. Concurrent navigation methods were lacking in several aspects, for instance they were not very precise and relied on good weather. This and the nuclear threat race provided justification to fund new and fairly costly pioneer technologies based on determining receiver location using a radio signal received from a set of space satellites. This is how the Global Positioning System, which is mostly known by its acronym GPS, was born. This system, later released for civilians use, works using a set of 33 satellites orbiting Earth and transmitting radio signal at high frequencies. The signal, carrying digitally coded transmission time and satellite position at the moment of dispatch, is used to calculate the receiver's location on Earth's surface. This localization system is fairly reliable and is being successfully used in many areas ranging from military applications through outdoor navigation and clock synchronization (with error in magnitude of hundreds of nanoseconds [2]) all the way to radio occultation [3] and tectonics [4] and was built upon and improved in several other next generation navigation systems, such as Russian GLONAS or European Galileo [1]. The two microwave radio frequencies used by GPS at 1.57542 GHz and 1.2276 GHz have however one very important disadvantage: they require fairly clear view of the sky, since they absorb very well onto a lot substances and materials, which means the technology cannot be used very well indoors. Prototypes of other localization technologies exist but a lot of them either have the same problem as GPS, require hardware support being integrated into the building, are not precise, or are currently in early stages of development. Another fact that must be taken into consideration is that logically most indoor spaces are fairly small and trying to use navigation tools in them may be somewhat counterproductive

as opposed to a bit of deduction and/or brute force search. These two reasons lead to the fact that there are very few systems available for navigation in indoor locations. However, some places, like the CTU Dejvice complex, can cause a lot of headache due to their maze-like architecture and their visitors may welcome tools that would help them finding their destination located inside. Some pioneering indoor navigation systems might be deployed for such use, however during my research I have found only commercial solutions that are targeted at large deployments and may require significant investment to implement; none of them open solutions readily usable in public places such as schools and hospitals.

1.2 Old CTU Navigator problems

Inside CTU, a long-running project is being worked on that aims to fill this vacant spot. This project, called the CTU Navigator, has already had two previous versions. The first version is based on dissertation of student Ondřej Čermák [5], who designed its client-server architecture. The second version was subsequently developed by an entire team of students who built upon the first version. These previous versions were designed to trace paths to desired rooms and were prepared for extensions that would make traversing these paths easier, however all ran into various fundamental roadblocks which averted their production use. The second version of CTU Navigator for example ran into problems with non-working server part and our team consultant Michal Maněna has also pointed out that it used raster images for its map grid, which introduced severe problems during its deployment. Raster images have the unfavorable disadvantage of having relatively large size when to be used in smartphone software. This combined with the design decision to support offline mode on mobile client apps integrated within the system meant that the target smartphones would have to download and keep packs of around 300 megabytes of various data and images, making the whole solution unsuitable for any practical use.

There are two immediately obvious solutions to this problematic design. One is to make the applications always on and continuously download map tiles similarly to how outdoor navigation systems like Google Maps work [6], or reduce the images in size in some way, such as converting these map images into vector formats that are usually smaller in size and then even further compressing them. The first way of continuous connection has two problems. First, it results into strain on potential user's mobile data plan, which may deter certain target audience from using the system. Second, mobile data connection may not be available in some deployments, especially inside concrete complexes without accessible WiFi connection. These two problems make the second approach to solving mentioned problem by reducing size of the map images a more viable solution.

All these circumstances led to the development of third version of the CTU Navigator project, on which I participate as a member responsible for the backend part and which my thesis is focused on.

Chapter 2

Analysis and design

2.1 Map data and client needs

As mentioned in previous chapter, preceding versions of the project already tried few unsuccessful design approaches. These undeployable prototypes have however led to formulating several new requirements for the next version, two of them being

(a) the client must have access to full offline data pack containing all necessary files required for full navigation function,

(b) map data must be supplied in compressed or vector map format, such as GeoJSON, to reduce their size.

The required map file format types are governed by available software on client platforms and were laid out by responsible team members.

Windows Phone and web For Windows Phone and web clients, Stanislav Mikeš and Jevhenij Horvat have in tandem chosen the OpenLayers v.3 library [7] [8]. This library can handle number of map file formats, from which we have decided to use the GeoJSON format known for its simplicity and widespread use.

Android Peter Janička has based the Android app client on a library called MapsForge, which mandates that the maps are provided in its own specific format. I have tried to locate available methods to convert data into this format, however there seems to be only one way to do that (apart from writing my own utility): OpenStreetMap project's Osmosis tool (hereinafter referred to as OSM Osmosis) in conjunction with the MapsForge Map Writer plugin. OSM Osmosis is a command-line utility developed under the OpenStreetMap project designed for processing various types of the project's data, mostly maps and databases. It has modular and pluggable architecture, into which

the MapsForge Map Writer plugin is integrated. The plugin is part of the MapsForge library, benefiting from OSM Osmosis map handling routines to convert various map file formats into the MapsForge project's own map format.

Osmosis being a command-line java utility precludes any easy integration with the MODX framework due to different types of their underlying technologies. One way to run the conversion would require outside calls from MODX (and thus PHP) to the operating system shell, which has traditionally been a dangerous endeavor from security standpoint. Another way would be running it from within an operating system-specific time-based job scheduler or filesystem watcher, such as Cron on Unix-like operating systems. Since both methods are equivalent in their result and since this is a decision that a system administrator deploying the software would be more competent to answer, I have left it unresolved.

Due to mentioned problems I have decided to convert all input map images into GeoJSON first and base all of my following work on this format.

2.2 MODX CMS

One of my tasks was to build the server backend part upon the Content Management System (CMS) software called MODX [9]. This CMS contains variety of features focusing but not limited to administration of content, user management, access control and others. It consists of two parts, runtime part handling presentation of stored data to potential users and administration GUI part called the Manager designed for management of most of its features and content. The whole CMS is designed mainly for deployment and management of semi-static web pages and web portals, less so for development of HTTP APIs, which was a large part of my work on the CTU Navigator backend.

2.2.1 Data structure

Resources The CMS uses fragments of text, called Resources, which are intended to contain most of dynamic content and which can be made public and directly accessible by clients via the HTTP protocol. The contained text may be anything but most of it is expected to be HTML as the prepared wrapping template for injecting standard HTML header and footer suggest. When a page is requested via HTTP request, a process is started which injects various fragments of data into the Resource's text based on markup annotations placed inside. These fragments of data are split into several types, called Template Variables, Snippets and Chunks.

Template Variables The Template Variables are basically fragments of data specific for given Resource. Each Template Variable has two selectable display methods which govern how their content is rendered on different occasions. The first one is Input, which defines how the data is displayed and

edited in MODX administration interface and is, on the lowest level, just collection of some HTML, CSS and JavaScript code. Second is Output that consists of fragments of PHP code and defines how the data is rendered for injecting into the generated Resource content. These Template Variables are divided into categories called Templates and each Resource has exactly one Template assigned.

Snippets The Resource content may be further augmented by Snippets, which are actual PHP code fragments that are run during the content generation and produce most of the dynamic parts, for example current date or user login name. Most of my API design and development is linked to working with these Snippets.

Chunks Third type of injectable data are Chunks. Chunks are just strings of plain text and mainly serve to produce different text versions for different languages.

Plugins Last content entity built into MODX are Plugins. Plugins are another chunks of PHP code but they do not interact with the content directly. They are rather run using several system hooks that are triggered on various system events such as saving a Resource definition, changing a Template, clicking on one of the Manager's administration menus, and many other.

Contexts To isolate logically different sets of its entities, MODX introduces Contexts. Each Context is basically a separate tree containing any of the previously mentioned MODX entities. Access rights may be also managed on the Context level, making them ideal to create categories based on languages, content versus API, site structure versus articles and so on.

Extras A completely separate MODX feature are Extras. These are collections of PHP, HTML, CSS, JavaScript and various other types of files that have one purpose: to extend core MODX functionality. They usually do not directly define content but rather help with its creation and transformations and are deeply integrated into MODX architecture. Various Snippet, Chunk and Plugin text editors, special input/output data formatters etc. fall into this category.

Documentation The MODX website contains quite a lot of tutorials ranging from simple introduction to administration to quite complex directions targeted at helping with implementation of new Extras. These tutorials are well organized but since the CMS developers seem to often change their parts during its evolution, some of them are already outdated at the time of writing this thesis and lead to non-working code. Extensive class reference can also be

found but since it is automatically generated and a lot of the public routines are not documented, it is severely limited and the routines still require quite a lot of reverse engineering to find out what they actually do. Many of them also seem to have no effect, making me speculate that they were deprecated yet not removed.

Internal structure Internal MODX structure, how it preserves the data and files and how it handles them is quite complicated in my opinion. Because the main CMS' goal is to ease management of content, all of the described objects are kept in the database (including code Snippets), which is accessed via an ORM mapper and which may be of one of several supported database types. The ORM mapper is not very well documented and to circumvent it is quite difficult due to inaccessible database connection parameters, which makes storing data in MODX possible only via one of the described content objects. These objects are stored into the database with a lot of extraneous data, such as creation and modification timestamps, the object hierarchy and others, causing a massive overhead when storing simple entities.

The CMS design does not seem to follow the Separation of mechanism and policy principle. While this principle originates from development of operating systems [10], it is easily applicable to almost any software development. It states that support routines in software that are used to provide certain aspect of its function, or mechanisms, should not directly dictate the process of how they should be used, or policy. I quickly ran afoul of this during experiments with implementation of my extensions to the CMS, when I tried to add code for uploading maps from one of my applets onto the server. I found out that to use MODX' authentication and authorization routines, I had to define a lot of new support classes sprinkled around several files. This is not a common pattern as in a lot of other frameworks and libraries one usually only has to add 2-3 lines of code to implement it. After the eighth implemented class, I had to ditch that attempt due to lack of time and instead leave the upload up to the user via default upload box in the Manager.

The Manager frontend is built upon a JavaScript framework designed for creating responsive GUIs called Ext JS, which is quite massive in itself and has steep learning curve. Integration of this framework into MODX violates the separation principle as well, making creation of any new HTTP API endpoints for use in the Manager very difficult and basically forcing any new developer without enough time to reverse engineer the solution to instead make use of the existing routines, accessible via the Ext JS. As mentioned above, the framework has steep learning curve, so the whole process is not an easy task.

This, combined with lacking documentation, makes any development beyond simple use of MODX content entities quite a laborious chore.

2.3 Navigation graph

2.3.1 Introduction

To facilitate navigation, providing only map data would be insufficient because they contain mostly entity data, such as placement of buildings etc., but seldom add information that can be used to actually plan walking route and when they do, it tends to be incomplete.

Navigation thus requires an additional structure called the navigation graph. For the backend part, I have named the graph's vertices waypoints and edges routes. I also needed a way to link positions of actual topographical entities, such as buildings, to the graph so when a user of one of the system client applications searches for a way from his original point to a point named "CTU FIT Elevator" the system can find out which waypoint is actually named as such. For this purpose I introduced a third entity type into the graph to solve this problem. I have named it PoI, shorthand for Point of Interest. The PoI is directly tied to a single waypoint and extends it with additional information, such as identification name and foreign routes. The graph must be split into several parts for administration reasons. Each part usually represents a building floor or an outdoor campus area and the foreign routes serve to sew them together by referencing two remote PoIs, each in a different map.

When building the graph, one has to define all of the waypoints, routes and PoIs. These are usually quite numerous for each graph part and doing it in MODX is not an option as it would very quickly become unmanageable due the sheer number of required entities. I have not found any existing solutions that would help with this task, so I have after discussion with my consultant decided to create a separate JavaScript applet for it that would store the resulting graph as a JSON-encoded blob using MODX' Template Variables. The resulting graph would be then connected to other parts, actual buildings and other geographical entities by its PoIs and foreign routes. I have named this applet `wayEdit` for easy differentiation from other applications created within this thesis.

2.3.2 Graph structure

As said, the navigation graph is to be stored and transferred in JSON format. The graph has two forms: an internal form, in which it is split into sections each bound to a specific map, and an external form, which consists of several files each containing one entity type of the complete graph. These files are packed into a zip archive and provided to a client upon a request. The internal form is described below, the external form in chapter 2.7.5.

```
{
  "waypoints": [
    [
      14.394224882125854,
      50.100796739834735
    ],
    [
      14.393130540847778,
      50.10203546464578
    ],
    [
      14.388796091079712,
      50.10416874895715
    ]
  ],
  "routes": [
    {
      "points": [
        1,
        0
      ]
    },
    {
      "points": [
        2,
        1
      ]
    }
  ],
  "pois": [
    {
      "point": 0,
      "name": "Metro: Vítězné náměstí",
      "foreign": [
        {
          "map": "karlak",
          "poi": "Metro: Karlovo náměstí",
          "cost": "3000"
        }
      ]
    },
    {
      "point": 2,
      "name": "Stavební fakulta",
      "foreign": []
    }
  ]
}
```


Figure 2.1: Example: Navigation graph, internal form.

2.3.2.1 Internal form

JSON file containing internal form of the navigation graph is shown in figure 2.1. All its entities are described below.

Waypoints The `waypoints` element contains a list of geographical coordinates each defining a single waypoint.

Routes All routes between two waypoints both contained in the same graph section are listed within the `routes` list. The routes have up to two children: `points` containing two indices into the `waypoints` list and optionally `cost` giving the route cost in meters. If `cost` is absent, the route cost is automatically calculated as Euclidean distance between its waypoints.

PoIs Finally the `pois` element contains list of Points of Interest. Each such point contains several children:

`point`	an index into the `waypoints` list
`name`	a human readable name of the point
`foreign`	a list of foreign routes

Each foreign route object has several children:

`map`	an identifier of the other foreign route map
`poi`	a name of Point of Interest identifying waypoint in `map`
`cost`	route cost (mandatory for foreign routes)

2.4 Images to map

Most of the clients of the CTU Navigator project use various libraries that allow them to load maps in one or more geo-related formats. As we need to work not only with outdoor areas but also with indoor ones such as building floors, we need to convert the related source materials at hand into these geo formats. Sources currently at my disposal are digital architectural plans in the PDF format. The conversion process consists of several steps:

1. Convert sources into digital images.
2. Convert these images into a predefined intermediate vector image format, such as SVG.
3. Perform cleanup to remove extraneous lines and other graphical artefacts.

4. Find out the geographical coordinates of several reference points in the image.

5. Convert the SVG image into desired geo-related format, such as GeoJSON, using the reference points from previous step.

6. Post-convert the map from GeoJSON into several other needed map formats as mandated by the client software.

The following sections focus on each of these steps separately.

2.4.1 Converting vector images

The input images may be supplied in any of number of formats that generally fall into one of two categories: vector images and raster images. Translating vector images into another intermediate vector format is fairly easy and there are number of tools at hand that can be used for this task. I have for example used the `convert` command of the ImageMagick project, which is a quite versatile image conversion tool capable of handling several different formats. Another tool I found suitable for this task is the Inkscape editor, which is directly targeted at editing SVG images and is capable of importing handful of other formats. These two are by no means an exhaustive list, there are many more ways to convert vector images but since these cover most commonly used formats, I leave it at that.

2.4.2 Converting raster images

Converting raster images into a vector format is way more difficult than converting vector-to-vector or vector-to-raster due to their fundamental conceptual difference. The vector images define lines, rectangles, circles and other geometrical objects by sets of mathematical equations that, when plotted onto a two-dimensional plane, translate into the shape of these objects. Raster images on the other hand work directly with this resulting plane, by using a two-dimensional matrix sample which they just compress in some implementation defined way to reduce its size. This leads to the fact that vector-to-vector conversion is fairly easy as it usually means just rewriting the mathematical equations from one notation to another or approximating them by another equation system native to the other format if that cannot accommodate the original. Vector-to-raster is also fairly straightforward, as it is just plotting the equations onto the two-dimensional point matrix. Raster-to-vector is however an entirely different story.

As vector images comprise of mathematical equations that define various geometrical objects, to convert raster images into vector one has to find equations that, when plotted, give at least similar look to the original. This is a complex and difficult task that has fortunately been researched before. Several

methods to trace edges in the raster image have been developed, one of them is for example the Sobel filter [11], that when incorporated into an conversion algorithm results in an vector image that resembles the raster original.

There are various tools that incorporate the Sobel filter or similar techniques to convert raster images into vector formats. Most of them are, unfortunately, either paid or online and those I have left out, paid tools because of their price, online tools due to their unreliability: an offline utility will work indefinitely but an online tool is dependent on its author's or host's support, should they decide to take them down, they become unavailable to the general public.

There are still several open tools for raster-to-vector conversion. First notable one is the `ImageTracer` utility [12], available in Java or JavaScript implementations. Another tool usable for this goal is `Potrace` [13], which has been directly integrated into the aforementioned Inkscape editor in its `Path->TraceBitmap...` command. The last free tool I found is WinTopo, a paid tool that at the time of writing this thesis has a freely available freeware version. All of these tools can convert into the SVG vector format.

An entirely outstanding approach to the conversion consists of using the Inkscape editor to import the raster image as a reference, manually drawing the vector representation over it and deleting the raster image afterwards. This is a time consuming process and a live user has to perform this task but it has at least two advantages. First, it also eliminates image cleanup. Second, one can draw straight lines. This results in much better vector images than most of the mentioned conversion tools produce as they smooth the traced edges and may create curvy walls and similar problems. Converting the image this way may be the most favourable approach for images of relatively simple structures with a lot of background noise and extraneous objects.

2.4.3 Image cleanup

After the image has been converted to intermediary SVG, an image cleanup is in order. Architectural plans depict many more objects than just walls, doors and the like, they often include fire barriers, measurements, legends and other unwanted items that clutter the image and often drastically increase size of the resulting vector image. Raster-to-vector conversion methods often add up to this by producing unwanted tracing artifacts. To clean these up, I have thought of two methods.

First cleanup method automatically filters tags of the SVG format using a script to find artifacts matching certain pattern and removes them. This method is imperfect as it usually recognizes only certain artefact types defined by the script author and their effectivity heavily depends on the conversion tool used beforehand.

Second cleanup method consists of plain manual cleanup using an SVG editor. I have tried this approach with the Inkscape editor and it provided

good results, albeit taking a lot of time up to about two hours for larger and more complex plans.

Most fruitful approach would probably be combining both methods by first filtering the most common and most easily machine-recognizable artefacts and automatically removing them and erasing less systematic artefacts manually afterwards. When trying it out on architectural plan of the ground floor of CTU New building, I was able to reduce its footprint from about 5 Mb to about 200 Kb, a considerable decrease.

2.4.4 Fixing reference points geographically

Basically all image formats use coordinate systems arbitrarily defined during the format conception. This is in direct contrast with most geo-related formats, which usually use one of the well defined geographic coordinate systems, most commonly latitude, longitude and elevation. To convert from our intermediary SVG into GeoJSON, we need to find out how their coordinate systems compare and the translation, rotation, scaling, and skew parameters needed to transform between them. To find these out, we need to set out several reference points on the image and find out corresponding geographical coordinates. The number of reference points depends on the used algorithm and therefore by proxy on used software. I have not found any tool appropriate for this purpose that would be sufficient for my needs and so I had to write my own. The resulting version uses exactly three reference points.

The three needed reference points may be located anywhere on the image but they must not lie on a straight line and lines drawn between any two of them should form as big an angle as possible to reduce conversion distortion. To find out their geographic location, it is usually sufficient to place an accurate GPS receiver at the place they depict. For this purpose one should choose features that stand out both on the image and physically. Building corners are usually good choice and they also commonly form an orthogonal grid.

2.4.5 Converting SVG to GeoJSON

I have found several utilities that already handle conversion from SVG to GeoJSON, unfortunately each of them displays various deficiencies making them unfit for my purpose. First such tool is the `ogr2ogr` command of the GDAL library, however this command can convert only specially formed SVG that has the geospatial data ingrained in its structure and thus cannot be used for my needs. Another one I looked into is the `dxf2kml` online application, which I had discarded because of its online nature (the reasons are the same as mentioned in chapter 2.4.2). Last library I found for this task is aptly named `svg-to-geojson`, however it uses only two reference points, meaning it cannot handle skew properly and also these two points are fixed at upper-left and lower-right corners of the image, which is not really convenient and

may require recalculation using more suitably chosen ones, meaning it would not significantly reduce the amount of needed work.

Because of described deficiencies and on account of relative simplicity of SVG and GeoJSON, I have decided to implement another JavaScript applet for this purpose. The applet would allow specifying three reference points needed to calculate required translation, rotation, scaling, and skew matrices and would directly perform the conversion. The JavaScript language was selected for convenience, firstly because it simplifies GUI design and implementation as several libraries exist that handle display of maps, secondly using the same language as for `wayEdit` would allow me to share some code between the two applets, thirdly all major browsers already have SVG parsers integrated and fourthly they also contain parser and serializer for handling JSON.

I have named this SVG-to-GeoJSON converter `svgGeo`. The conversion process is a multistep algorithm using fair amount of linear algebra, which is described below in following section.

2.4.5.1 Structural differences

Both the SVG [14] and GeoJSON [15] are well standardized and widely used formats. Both are structured into a tree of elements which makes conversion from one to another fairly straightforward. There are only two significant differences in what they can graphically and topographically represent. First difference lies in the fact that GeoJSON does not work with color, while SVG does. GeoJSON-formatted maps are usually after-colored by the display software using additional property data embedded in its structure. These additional properties are however not standardized and thus the coloring is very specific to the rendering software and may differ from one map to another or may be completely missing for some of them. SVG on the other hand has full color support. The second difference is that GeoJSON works only with straight lines, while SVG supports definition of circles and Bézier curves, which are curves described by Bernstein polynomials [16]. Third notable point, more a SVG specialty than a difference, is that the SVG can contain post-processing spatial transformations which can successively transform parts of or even whole image. The first structural difference does not matter much as we can use the whole embedded source-specific property data to color outdoor maps while indoor maps are so small that they do not need varied coloring to make them readable, the second difference means that the circles and curves must be approximated by implementation-defined count of straight lines. The third SVG quirk in being able to post-transform the image meant more work to implement these necessary features but having to analyze this feature proved advantageous for me later on.

2.4.5.2 Coordinate system differences

As has been hinted previously, the coordinate systems used by SVG and GeoJSON differ in several ways. First difference is in how their origin point is defined. The origin point of SVG is more or less arbitrary and does not set any strict boundaries since SVG coordinates can be negative and the whole image can be post-transformed as mentioned in the previous section. On the other hand GeoJSON does not have any such capabilities, so all of the SVG coordinates must be processed to yield absolute values and those have to be precisely translated afterwards. Second notable difference lies in the fact that the GeoJSON coordinate axes are finitely bound with the same limits as latitude and longitude, meaning that corresponding coordinates must fall into $[-90, +90]$ and $[-180, +180]$ intervals respectively. Because SVG axes scale infinitely, this leads to a necessity of also having to scale the coordinates by some image-specific scalar value to make them fit into those bounds. Moreover SVG Y and GeoJSON latitude axes have opposite orientation, so the sign of all Y coordinates must be flipped during conversion.

Last important fact that should be noted is that the coordinates fundamentally differ in what they map onto. The SVG coordinate system is planar, while GeoJSON coordinates naturally map onto an ellipsoid. Precisely converting from one format to another would thus require a map projection and fairly complex mathematical calculations that go hand in hand with it. Still, the CTU Navigator system works with maps of fairly small areas, a single kilometer in any direction at most. Maps of so small areas would suffer negligible distortion if the conversion was greatly simplified into just linear transformations that would ignore the Earth's ellipsoid nature, as between two planar coordinate systems. This insignificant distortion cost finally led me to using linear transformations as the basis of my conversion design.

2.4.5.3 Conversion steps

The conversion process from SVG to GeoJSON consists of several steps performed in an order given partly logically and partly by chosen technologies. The steps go as follows.

1. Parse input SVG file to yield structural elements.
2. Traverse the resulting tree, converting each element into a GeoJSON feature with coordinates transformed to fit in GeoJSON' geographic coordinate system.
3. Render the resulting map in GUI and allow setup of the reference points.
4. Retrieve the reference points' values and use them to calculate a matrix representing necessary translate, rotate, scale, and skew operations.

5. Multiply all GeoJSON coordinates acquired in step 2 with the transformation matrix.

6. Serialize the GeoJSON feature tree into output GeoJSON file.

2.4.5.4 The math

When analyzing and preparing the mathematical groundwork needed for calculation of the transformation matrix, I have greatly benefited from analyzing SVG transform operations beforehand. As I mentioned, the SVG specification defines a series of transformation operations used to alter its image parts before rendering. These operations, their matrices, and variables in order are:

Translation

$$\begin{pmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{pmatrix}$$

Translation shifts all image point coordinates by fixed amount for each axis. x represents shift distance along the X axis and y distance along the Y axis.

Scaling

$$\begin{pmatrix} x & 0 & 0 \\ 0 & y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

When scaling, all coordinates are translated by progressive amount dependent on their distance from the pivot point at $(0, 0)$ and on two scaling factors. Here x represents X coordinate scaling factor and y Y coordinate scaling factor.

Rotation

$$\begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Rotation uses the same pivot as scaling, rotating the image around it with α representing the rotation angle.

Skew

$$\begin{pmatrix} 1 & \tan\alpha & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} 1 & 0 & 0 \\ \tan\beta & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Finally skew represents a skew operation along a specified axis. The shift distance in given direction is dependent both on absolute value of the relevant

coordinate and on skew angle. In the matrices above, α represents skewing angle for skews along the X axis and β angle for skews along the Y axis.

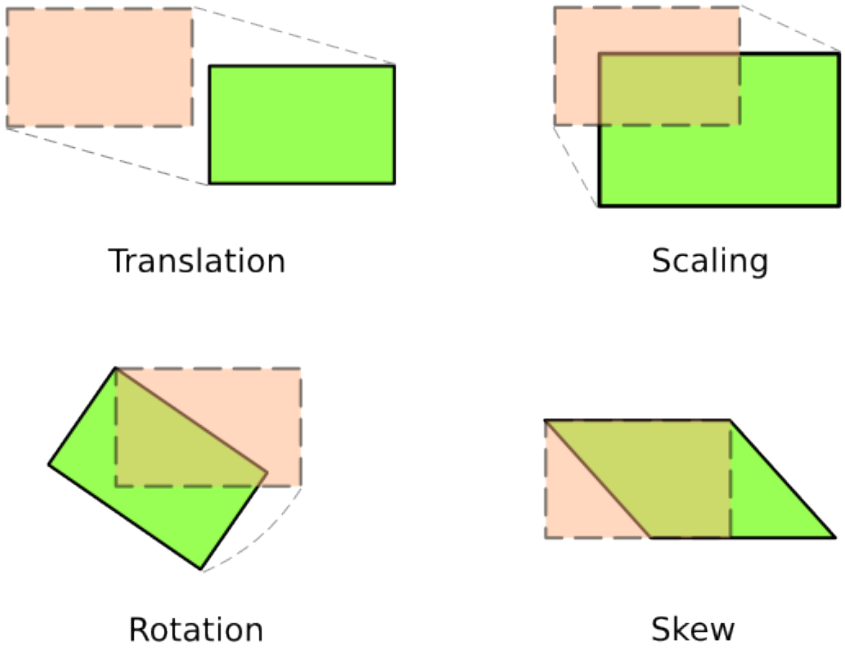


Figure 2.2: Transformations

Depictions of all of the described transformation types may be found in figure 2.2.

Transformation matrix calculation algorithm To produce the final transformation matrix, we calculate each of the four operations using differences of the reference points and their geographical positions. The three reference points each play a strictly set role for the duration of the conversion, yet they may be given those roles fairly arbitrarily at the start as long as the conditions linked to their selection hold. The roles in question are that of a pivot point, a scaling point and a skewing point, hereinafter referenced by symbols A, B and C. Each point actually comprises of two, separate, positions, one in the image, one geographical. The image ones will have subscript $_i$ and the geographical ones $_g$. So all in all we will be working with six symbols: $A_i, B_i, C_i, A_g, B_g, C_g$.

The algorithm based on analytic geometry that moves the image coordinates to their geographical position is composed of following steps.

1. Translate the image so that $A_i = \binom{0}{0}$, or that the pivot point really becomes the pivot for following transformations.

2. Rotate the image so that $B_i = \binom{n}{0}$, $n \neq 0$, or that vector from the pivot point to the scaling point lies on the X axis.

3. Skew the image along the X axis so that C_i and B_i form the same angle as their geographical counterparts.

4. Scale X coordinates of the image so that X coordinate of B_i equals to $|B_g - A_g|$, or that the vector from the pivot point to the scaling point has same length as its geographical counterpart.

5. Scale Y coordinates of the image so that Y coordinate of C_i equals to $|C_g - A_g|$, or that the skewing point has same distance from the X axis as its geographical counterpart.

6. Rotate the image back so that the B_i and B_g have the same direction, or that the image is rotated into its final geographical angle.

7. Translate the image so that A_i lies at A_g, or that it lies where it should lie.

Translating this algorithm to a set of mathematical equations to produce the transformation matrix yields

$$A_i = \begin{pmatrix} x_{ai} \\ y_{ai} \\ 1 \end{pmatrix},\ B_i = \begin{pmatrix} x_{bi} \\ y_{bi} \\ 1 \end{pmatrix},\ C_i = \begin{pmatrix} x_{ci} \\ y_{ci} \\ 1 \end{pmatrix}$$

$$A_g = \begin{pmatrix} x_{ag} \\ y_{ag} \\ 1 \end{pmatrix},\ B_g = \begin{pmatrix} x_{bg} \\ y_{bg} \\ 1 \end{pmatrix},\ C_g = \begin{pmatrix} x_{cg} \\ y_{cg} \\ 1 \end{pmatrix}$$

$$V_{stab} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$$V_{bai} = \begin{pmatrix} x_{vbai} \\ y_{vbai} \\ 1 \end{pmatrix} = B_i - A_i + V_{stab}$$

$$V_{bag} = \begin{pmatrix} x_{vbag} \\ y_{vbag} \\ 1 \end{pmatrix} = B_g - A_g + V_{stab}$$

$$\alpha_{vbai} = \arctan(\frac{y_{vbai}}{x_{vbai}})$$

$$\alpha_{vbag} = \arctan(\frac{y_{vbag}}{x_{vbag}})$$

These are vectors from pivot point to scaling points and their directions. They can now be used to produce inverse rotation matrices that in turn can

be used to rotate similarly produced skewing to pivot vectors. This has to be done to get scaling factors so one can produce the scaling matrix.

$$M_{imgInvRotate} = \begin{pmatrix} \cos(-\alpha_{bai}) & -\sin(-\alpha_{bai}) & 0 \\ \sin(-\alpha_{bai}) & \cos(-\alpha_{bai}) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$M_{geoInvRotate} = \begin{pmatrix} \cos(-\alpha_{bag}) & -\sin(-\alpha_{bag}) & 0 \\ \sin(-\alpha_{bag}) & \cos(-\alpha_{bag}) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$M_{geoRotate} = \begin{pmatrix} \cos\alpha_{bag} & -\sin\alpha_{bag} & 0 \\ \sin\alpha_{bag} & \cos\alpha_{bag} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

This produced the rotation matrices, the first two will be used to rotate skewing vectors, the first and last one will be used to rotate the image to its proper orientation.

$$V_{cai} = \begin{pmatrix} x_{vcai} \\ y_{vcai} \\ 1 \end{pmatrix} = (C_i - A_i + V_{stab}) * M_{imgInvRotate}$$

$$V_{cag} = \begin{pmatrix} x_{vcag} \\ y_{vcag} \\ 1 \end{pmatrix} = (C_g - A_g + V_{stab}) * M_{imgInvRotate}$$

Now the scaling and skewing vectors can be used to calculate the scaling matrix.

$$s_x = \frac{|V_{bag}|}{|V_{bai}|},\ s_y = \frac{y_{vcag}}{y_{vcai}}$$

$$M_{scale} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

That is for the scaling matrix. We have to use it to scale the image skew vector and then use difference of the image and geographic skew vectors to give the image proper skew.

$$V_{cai} = V_{cai} * M_{scale}$$

$$f_{skew} = \frac{x_{vcag} - x_{vcag}}{y_{vcag}}$$

$$M_{skew} = \begin{pmatrix} 1 & f_{skew} & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

At this point the only thing that is missing are translation matrices, which are pretty straightforward to create.

$$M_{imgInvTranslate} = \begin{pmatrix} 1 & 0 & -x_{ai} \\ 0 & 1 & -y_{ai} \\ 0 & 0 & 1 \end{pmatrix}$$

$$M_{geoTranslate} = \begin{pmatrix} 1 & 0 & -x_{ag} \\ 0 & 1 & -y_{ag} \\ 0 & 0 & 1 \end{pmatrix}$$

And now to put it together.

$$M_f = M_{geoTranslate} * M_{geoRotate} * M_{skew} * M_{scale} * M_{imgInvRotate} * M_{imgInvTranslate}$$

Transformation The transformation matrix is used on the image simply by multiplying all of the image points with it.

$$A = \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix}$$

$$B = \begin{pmatrix} x_2 \\ y_2 \\ 1 \end{pmatrix} = A * M_f$$

where A is the original point, B is the transformed point and M_f is the previously calculated transformation matrix.

2.5 Third-party integration

I was also tasked with designing adaptors for integrating two third-party projects into my code, namely the OpenStreetMap project and Google Calendar. While the OpenStreetMap project integration was fairly straightforward, integrating the Google Calendar was quite harder due to its completely different nature. Each project is analyzed in its own following section.

2.5.1 OpenStreetMap

2.5.1.1 Introduction

The OpenStreetMap project, accessible at url `http://openstreetmap.org` at the time, is an open internet project shielded by the OpenStreetMap Foundation that aims to create and maintain up-to-date world-wide map data accessible under a free license. It is a fairly unique project since all other available mapping projects are usually maintained for commercial purposes and provide their map data freely usually only for personal use. The project's data is

kept and made accessible in several vector map formats including GeoJSON, making the project ideal for integrating into the CTU Navigator project as to provide for outdoor areas. The OpenStreetMap format also collects a lot of different metadata about objects stored in its database. For example the project discerns between highways, motorways, cycleways, pedestrian paths, which motorways have sidewalks for pedestrians, etc. This also gives maps coming from this project bonus value in that these metadata can be used to automatically build segments of the navigation graph for supplied outdoor areas, which only need minor finishing touches to be usable for full navigation.

2.5.1.2 Use cases

The project understandably only provides map data of outdoor areas due to variety reasons. The outdoor maps managed by the CTU Navigator project are expected to be comparatively few as opposed to interior ones, where every building and every floor will probably have to have its own separate map due to slight differences. The outdoor maps will however be mostly way bigger, as can be glimpsed upon just by looking at the CTU Dejvice campus in the OpenStreetMap website online viewer. The maps will be cached locally, making retrievals from the OpenStreetMap few and far between as the outdoor areas they depict usually seldom change.

2.5.1.3 Integration

The project does sport several APIs that allow access to the project's data in several ways depending on the intended use.

Editing API The lowest-level of these APIs is named OpenStreetMap Editing API. This RESTful API allows for fetching and saving raw data and is intended to be used, as its name suggests, mainly for editing existing data or loading new to the project. The map data is provided wrapped in an XML-based format and is not suitable for use with most of existing map rendering libraries or utilities without some serious reprocessing, making it unsuitable for integration into the CTU Navigator backend.

Overpass API Another interface to the project is the Overpass API. This is a read-only API built on a query language specific to it. The query language is designed to allow fetching specific subset of OpenStreetMap data based on several restricting criteria, such as location, object type, tag properties etc.

Third-party interfaces The project website also keeps track of quite a few software libraries and utilities developed in various programming languages and for various environments that cater for easy retrieval of the map data,

either split into tiles or raw, and reprocessing them into a form suitable for viewing.

Manual export Finally the project website allows for simple manual export of small areas in GeoJSON format via its online viewer. These exported files may then be directly loaded into the CMS and used.

Resolution After considering all of these options and intended usage, I have decided not to create any bridge between the two projects and instead leave it up to the user to export map segments via the OpenStreetMap website online viewer and manually upload them as files via MODX Manager. This should not represent any significant amount of work as the outdoor maps are expected to be few and the need to update them should rarely arise due to their nature. The maps are to be uploaded in the GeoJSON file format to a specific folder in designated for them. This is the same folder into which the indoor maps are to be uploaded and both types are afterwards handled in the same way.

2.5.2 Events and Google Calendar

2.5.2.1 Introduction

The Google Calendar, available at `https://calendar.google.com`, is built to aggregate descriptions of user-defined and third-party events, organize them into weeks and months and serve them in a variety of ways to several types of devices. It is for example directly embedded into the Android OS, with other mobile platforms having glue code and/or applications to access its data and incorporate it into their own time management applications.

2.5.2.2 Project's purpose

What was not instantly clear was how to integrate the project with the CTU Navigator backend, after all, the backend's purpose is to provide navigation and mapping data and serve them to its clients, a goal the Google Calendar does not really fit into. However after a quick discussion with my consultant it became clear that the backend was to become an all-around data aggregator in the future and this project was to be the first step in this direction. The school has all kinds of events going on, including promotion events, open days, student organized events, public talks, and other. Several methods used to keep a record of these are to be integrated into the backend and their aggregated collection is to be provided to its users. Integrating Google Calendar is a first step in this direction.

2.5.2.3 Integration

The Calendar provides an API using HTTP and JSON to access its contents via an authorization and access point token first retrieved using its web GUI. Since the backend is supposed to provide the aggregated events data in an iCal format as we agreed during one of our team meetings, it makes sense to retrieve the events via the API and immediately compile and store it as an iCal file, which can then be simply filtered and concatenated with other such files to provide desired output. The internal mechanism to do so is described in the section MODX structure design2.7.

2.6 Licensing

I use several third-party software libraries, tools and data in my work and these are provided under variety of open and free licenses. These licenses may preclude the use of resulting work or its parts from certain uses and/or without special approach to its use. The libraries and tools and their licenses in question are

1. MODX Framework, published under GNU General Public License version 2.0 (GNU GPL 2.0) [17]
2. OSM Osmosis tool, placed in public domain
3. MapsForge Map Writer plugin for Osmosis, published under GNU Lesser General Public License version 3.0 (LGPL 3.0) [18]
4. Leaflet JavaScript library, published under BSD 2-clause license [19]
5. OpenStreetMap project's map data, available under Open Data Commons Open Database License v1.0 [20]

Most of these licenses were analyzed under Czech Law by Matěj Myška et al. under patronage of Masaryk University in Brno [21].

2.6.1 GNU General Public License 2.0

The GNU GPL 2.0 is a widespread license which allows software in question to be used for any purpose, provided any modifications or anything that uses parts of such software must be made available to public under the same license. This is a fairly aggressive licensing that makes almost any linked work unsuitable for use in proprietary commercial solutions. In my case, I had to integrate my WayEdit plugin into the MODX framework. While the plugin itself is an applet written in HTML and JavaScript and does not integrate MODX code in any way and thus may be licensed under any license compatible with the BSD 2-clause license used by its Leaflet library base, the intermediary code

integrating it to the framework does and as such I believe this code has to be made public under the GPL 2.0 license if it is to be distributed.

2.6.2 GNU Lesser General Public License 3.0

From the same organization as the GNU GPL license comes its version specifically targeted for use with libraries, called GNU Lesser General Public License 3.0. It contains explicit clauses to allow such libraries to be used in any project without enforcing the copyleft principle on its other parts as long as that project does not modify or copy code of the LGPL licensed part in any way. Since I employ the MapsForge Map Writer plugin, published under this license, as is without any modifications, I believe this license would only affect distribution outside school, in which case a documented way to find the MapsForge Map Writer source and to replace it with another version would have to be provided.

2.6.3 BSD 2-clause License

The BSD 2-clause license is a derived work of an ancient license devised by the Computer Systems Research Group for use with their variation of Unix operating system, the Berkeley Software Distribution (BSD). It explicitly waives all rights to the software as well as any accompanied responsibility. It only requires the derived software to be supplied with this waiver. This makes it suitable for use in almost any kind of commercial or free software and allows it to be distributed under any kind of license, giving me a free hand to distribute the wayEdit and svgGeo applets under almost any license I would choose.

2.6.4 Open Database License v1.0

Tailored specifically for openly shared databases, the Open Database License aims to allow anybody to use any database licensed under it for any purpose, be it non-commercial or commercial, as long as they adhere to several principles. These principles have in mind a goal similar for data as the GPL license does for software, that is, any adapted version of such database that is publicly used must be made available under the same license and if such original or adapted database is to be redistributed, access to such redistribution may be restricted as long as the same version is also made openly available without any restrictions. Any public use of such licensed data must be attributed as the license specifies and must retain any accompanied notices. What this means is that the OpenStreetMap data provided under this license may be freely used in the CTU Navigator project, as long as they are clearly marked that they come from the OpenStreetMap project.

2.6.5 Software in public domain

Last but not least placing software in public domain means waive of all rights and responsibility to the software, without any conditions and, and this is important, without giving any kind of license to its users. This is problematic under the law of Czech Republic, since it does recognize such approach to the rights to a work, translated in Czech language as "volné dílo", only under a set of specific conditions, such as 70 years after the author's death. To reach similar effect the authors of the Osmosis tool would have to publish a special type of license called "free license for any use" to the general public, which they did not do, meaning they still retain all their rights and responsibilities to the software and could possibly try enforce those against any potential user in court under Czech law or vice versa. This is however highly improbable, since it would go directly against their original intent. I as a user of the Osmosis tool benefit from the academic nature of this thesis, since the Czech Copyright Act [22, article 35, point (3)] directly states that "Copyright is not infringed by a school or school-related or educational establishment if they use for teaching purposes or to meet their own internal needs a work created by a pupil or student as a part of his school or educational assignments ensuing from his legal relationship to his school or the school-related or educational establishment (school work), provided that this is not done for the purpose of any direct or indirect economic or commercial advantage.". This gives me an exception allowing me to use it for purposes of this work but in the case of further distribution outside of academic grounds this factor should be taken into consideration.

2.6.6 Making thesis' code public

All code written by myself that requires special handling due to copyleft licensing of integrated code was accounted for in previous chapters. I am as per our agreement handing all my ownership rights to the rest of the code to my alma mater, Czech Technical University in Prague. There is however one part that deserves special attention, the svgGeo conversion applet. Since I have not found any sufficient alternative that could handle its tasks and since I believe its purpose is generic enough to come handy outside of the CTU Navigator project, I have after consultation it with my supervisor decided to release it for public use under the BSD 2-clause license.

2.7 MODX structure design

2.7.1 Entities

Large amount of content managed in the backend requires careful planning of its structure. I have decided to separate the content types into several Con-

texts. First Context called `API` contains all of the dynamic API code, which is closely described in a chapter below. Structural data, such as entities containing parts of the navigation graph are held in Context named `Structure`. Lastly all of the textual data, such as building and PoI descriptions, information about school and others can be found in the `Content` Context.

2.7.2 API

The API Context holds all of the HTTP entry points via which clients retrieve data from the backend. The entire API was designed by the team to allow for differential updates. Its general outline of callable URL paths show in the table below.

`/version`	Data package version used to skip updates.
`/info`	Initial information with list of languages.
`/data-list`	List of files + fingerprints for incremental updates.
`/update`	Data package serving a zip archive with requested files.

In-depth API documentation is stored on the enclosed CD.

2.7.3 Structure

`Structure` Context holds parts of the navigation graph and other necessary data used to provide navigation. At its heart is a folder named `Planes`. This folder contains MODX Resources expectably having a Template named `Plane`. These resources each provide for various outdoor campus areas and building floors and hold corresponding navigation graph parts. These parts are stored in Template Variables bound to the Plane Template and Planes are then referenced by the `Content` Context entities via another set of Template Variables.

2.7.4 Content

In the `Content` Context lies a tree holding all textual information presented to the user. This tree is split into separate directories each corresponding to one provided language. When clients request data packages, they provide a language URL query option which is used to select relevant directory.

2.7.5 Content zip

At client's request a current or cached snapshot of all of the aforementioned Contexts is melded into one big zip archive. This archive contains several strictly defined files containing structural data and a variable file tree containing the "Content" Context. Layout of the archive can be seen in figure 2.3.

```
version.json ..... package data version (same as /version API call)
info.json ..... basic information, contains language list
data-list.json ..... list of files and their fingerprints
buildings.json ..... buildings list
planes.json ..... outdoor area or building floor objects
waypoints.json ..... list of navigation graph waypoints
routes.json ..... list of navigation graph routes
pois.json ..... list of navigation graph pois
assets ..... directory containing assorted data files
    maps ..... directory containing plane maps
    ..
images ..... directory containing images
icons ..... directory containing menu icons
    ..
```

Figure 2.3: Structure of the data update package

2.8 Workflows and diagrams

Handling the CTU Navigator backend is linked to several software and user processes. This section describes some of them and provides few diagrams, both for these workflows and a class diagram.

2.8.1 Workflows

Making a map usable Figure 2.4 illustrates the process of getting a map from variety of expected input formats into production. It shows that images are much harder to put in any usable stable state, since they require conversion, filtering and more.

Data package retrieval Client application has to perform several steps to get or update its data package. This process is illustrated in figure 2.5. The process allows for skipping if no relevant data have changed, both on package and on file level. It also shows how each API entry point fits into this process.

2.8.2 Class diagram

There are two areas of code that would normally warrant class diagrams within this thesis. One of them are the two JavaScript applets, `wayEdit` and `svgGeo`, whose class diagram is shown in figure 2.6. They share a single one since they also share fairly large amount of code. The other area that one would normally expect would be structured in object oriented way and as such should have a class diagram made for it would be the server HTTP API part, done in MODX. This is however not true, since MODX Snippets are pasted into other

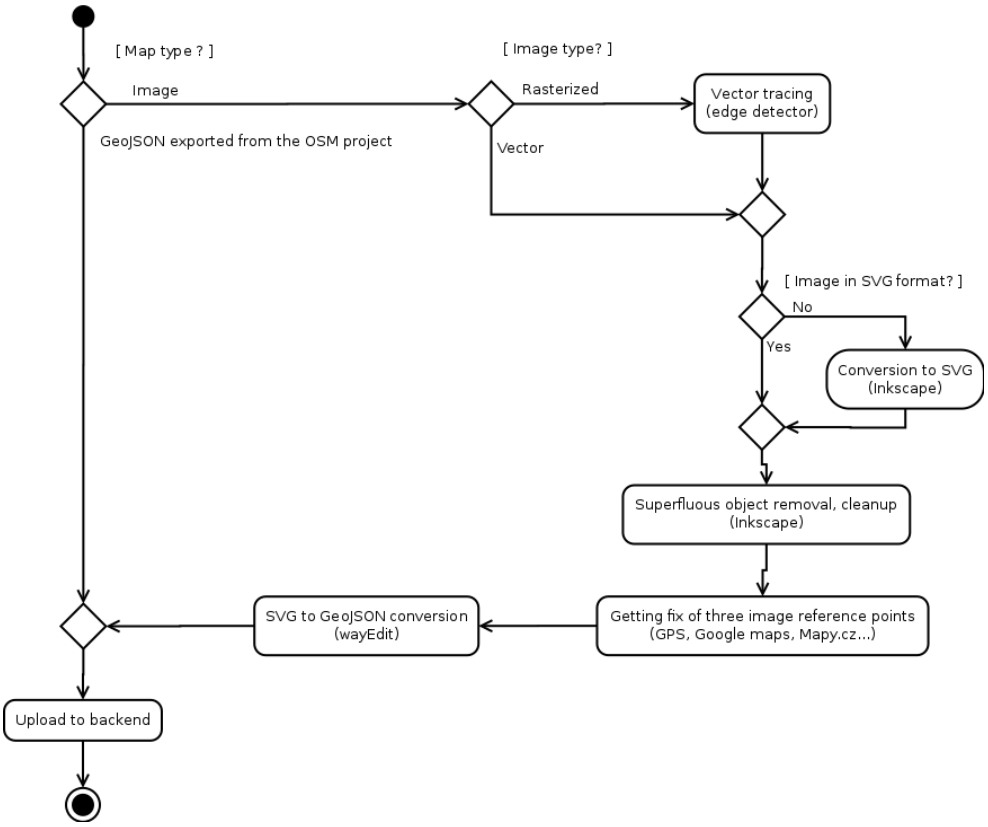


Figure 2.4: Workflow diagram: preparing map for production

MODX Snippets and into background MODX code in unspecified ways and PHP allows class and function definitions to be placed only at global scope. What this means is for the Snippets not to be fragile they have to refrain from using these two constructs and any complex code has to be structured by using just the Snippets. Thus there are no classes to create class diagram for in the HTTP API part.

2.9 FURPS

FURPS is an acronym that stands for Functionality, Usability, Reliability, Performance and Supportability. It is a model used to classify functional and non-functional requirements for a project. Since this thesis deals with a software project, a lot of categories, mostly non-functional, are moot since they are set out by used hardware and/or core technology, both of which are out of my control or set out in the assignment. The functional requirements were laid out by the thesis assignment and during team meetings by client needs, which I described in previous chapters. To reiterate them:

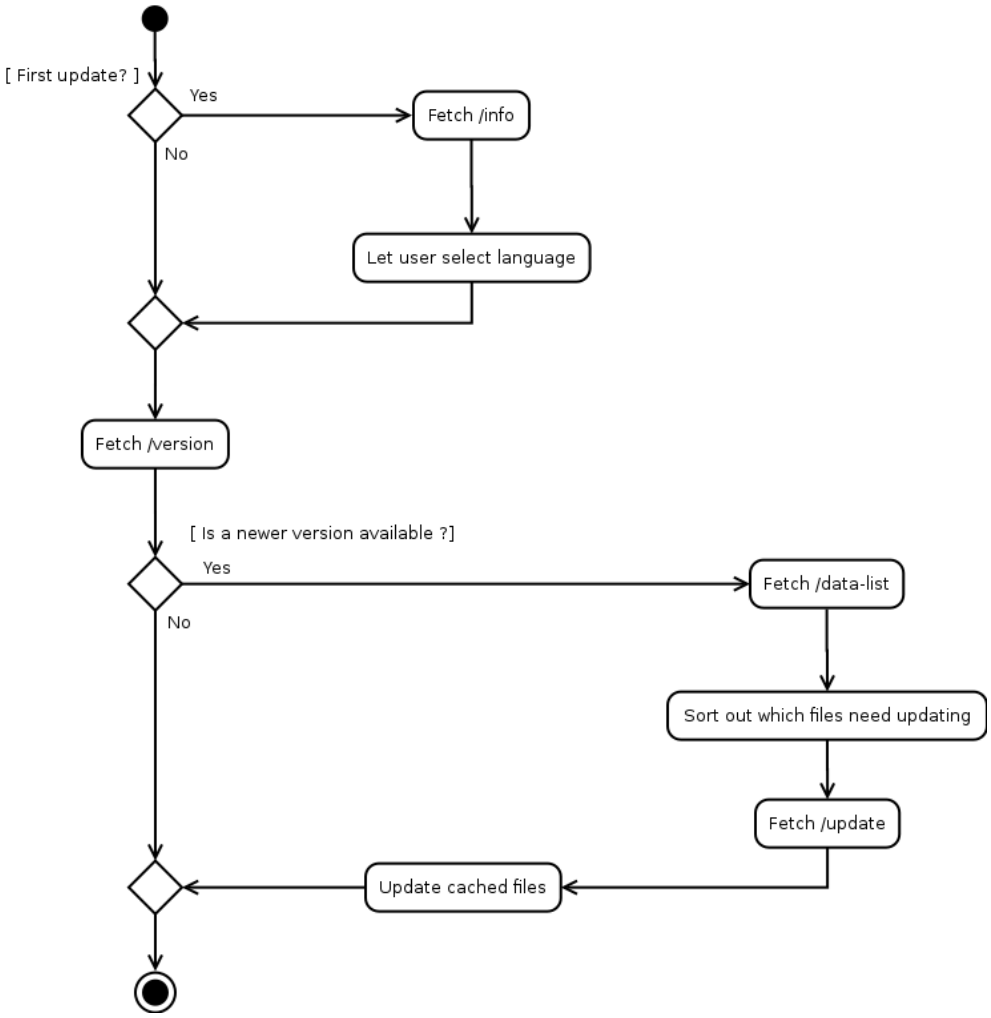


Figure 2.5: Workflow diagram: Client update

1. A way to convert images, such as building construction plans, into valid maps in requested map formats, namely GeoJSON and MapsForge .map, must be devised.

2. Administration GUI to handle content and navigation graph must be designed.

3. The GUI must allow to control access to administration activities.

4. Google Calendar and OpenStreetMap projects have to be integrated into the work.

5. Push notification support must be implemented.

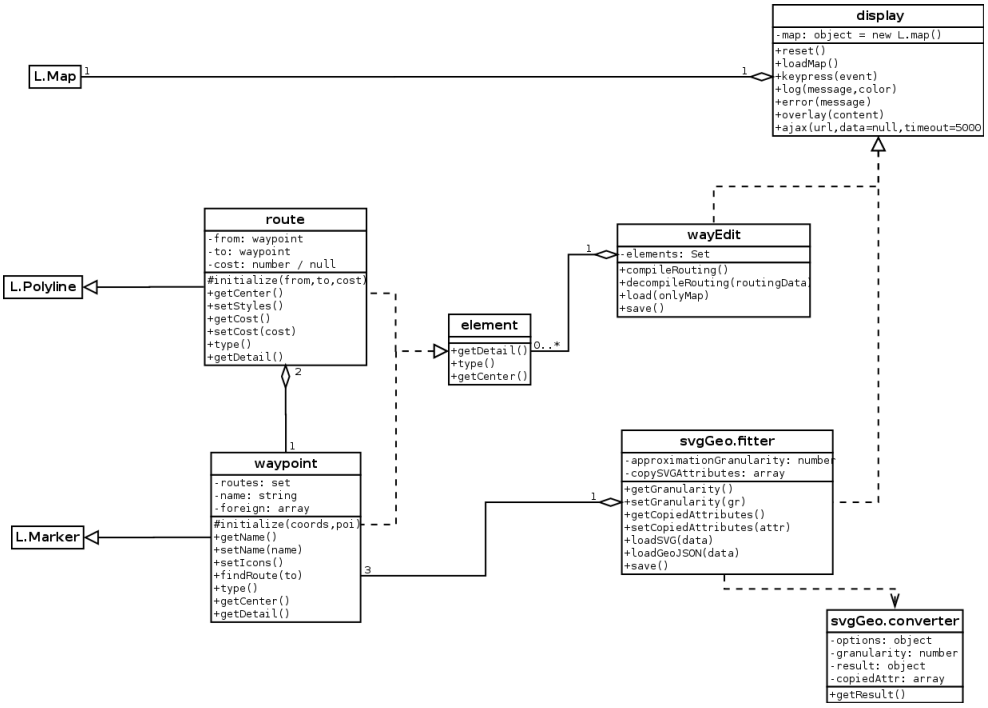


Figure 2.6: Class diagram: JavaScript applets

The only non-functional requirement that can be influenced by me in any way is system performance. No hard requirement was given, the goal is for the client applications to receive answers to queries in matter of seconds (sans round trip time) to be responsive to the user. This is already fairly offset by MODX, which is quite heavy-duty in this regard and consumes a lot of CPU time on its own.

Chapter 3

Realization

3.1 Development environment

3.1.1 Developing JavaScript applets

For development of JavaScript-based and partly PHP-based code I have used my standard development environment. This environment constitutes of the `Linux` operating system with the `Vim` editor adapted for my editing needs, a `Git` version control system for a separate repository set up for each component, Python with a module called `SimpleHTTPServer` for testing JavaScript applets during development and the `Firefox` browser for accessing remote MODX development instance to which I was given access.

My only entry point allowing me to administer the remote MODX instance it was the Manager, MODX' own inbuilt administration interface. This proved quite problematic as I did not have access to the PHP execution logs and other operating system-specific debugging tools. My only source of debugging data was MODX' own Error Log, which contains very limited amount of information mostly dealing with MODX specific exceptions and warnings and which proved quite useless for any practical use.

3.2 wayEdit navigation graph editor

To implement the `wayEdit` applet I had to provide adequate environment in which the user could see the navigation graph part and edit it on top of the map it is tied to. I researched available solutions and found a number of libraries for displaying maps. These libraries would require various amount of support code to serve my needs. I have pinpointed two that were most readily available for use named Leaflet and OpenLayers. They were most usable since they are written in JavaScript and are thus integrable into MODX. The OpenLayers library was later used by team members to cater for web client and Windows Phone platform, I have however decided to use the Leaflet library since it was

used by the OpenStreetMap project for its web map display and was thus guaranteed to work perfectly with outdoor maps as this project is integrated into the backend to provide them.

3.2.1 The Leaflet library

The basic Leaflet library architecture is based on a single display HTML element which allows adding a number of objects that define their rendering characteristics and the library uses them to display maps. These objects include paths, which are basically strings of straight lines, polygons, markers (points in map suitable for marking Points of Interest) and other items. The library also has routines for importing maps in the GeoJSON format, which were perfect for my needs. Another notable feature is ability to add various controls, such as buttons, menus and other.

3.2.2 Implementation

I extended two Leaflet objects, the polyline and the marker, to display two main navigation graph entities, routes and waypoints. To allow editing, I had to implement actions for selecting, creating, deleting, and drag-and-drop. Adding the objects to the graph is done via two toggle buttons, which, when on, cause mouse clicks to add the relevant objects to the map. All controls are both accessible by buttons and key shortcuts. The toggle method is fairly non-standard and is primarily designed with productivity and efficiency in mind rather than ease of use, which in retrospect might not be the best approach as it requires a bit of learning the controls to be able to efficiently use the applet. Nonetheless, once the user passes this phase, creation of graphs on larger maps should be a fairly short affair.

3.2.3 Integration into MODX

Integration of this applet as a Resource Template Variable input filter in MODX Manager proved quite challenging. MODX uses the Ext JS framework and is tightly integrated with it. Both of these impose fairly complex policies on integrating extensions into them. These policies are at least in case of MODX not very well documented and made me attempt to circumvent them by copying most of the integrating code from one of the existing inputs and bending it to suit my needs. To display the applet, I completely evaded Ext JS display objects and used MODX to paste some HTML code in place of the Template Variable and working on top of that. I am aware of several problems this causes, for example the user editing the graph has to save it twice, first in the applet to store the serialized data in an hidden input HTML element and second in Manager to save contents of this hidden input element into its database. Another problem that appeared was in

Ext JS' method of separating elements in tabs and displaying one tab at a time, which broke Leaflet initialization and made me provide a button for completely reinitializing the applet. This would not be easy to fix even if the applet was integrated properly as I suspect it stems from the display element not having width or height since it is not located at the first tab which is displayed on load. All of these issues are fairly serious, yet to fix them would require studying the Ext JS leaflet in depth and reverse-engineering MODX since its documentation is insufficient to devise a proper solution. In future I would advise changing core technology of the backend from MODX to a more suitable framework or library, because even though I understand the benefit of MODX' authentication routines, extending it is quite hard as compared to alternative and more ubiquitous solutions.

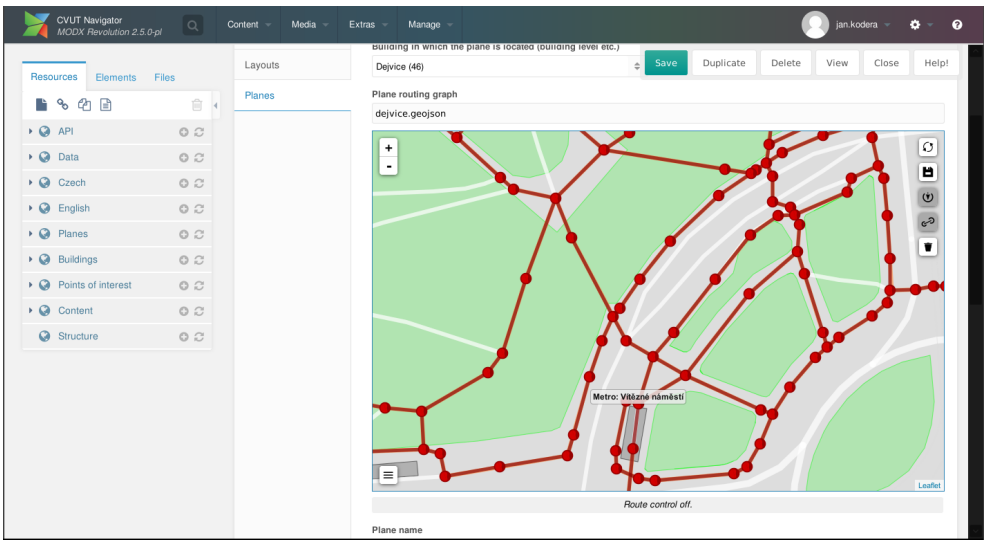


Figure 3.1: wayEdit integrated into MODX Manager

An illustration of the applet integrated into MODX may be seen in figure 3.1.

3.3 svgGeo conversion applet

To implement the `svgGeo` applet I leveraged codebase of the `wayEdit` applet and used its waypoints and menu for reference points and load/save actions respectively. The code is fairly straightforward, with conversion from SVG to GeoJSON happening early during load operation to allow loading by the Leaflet library and translating the map to correct coordinates during save. The algorithm after some amount of debugging works as expected for maps of small enough areas, which goes in line with its intended use.

Only real problem during its implementation again arose during my attempt to integrate it into MODX. I wanted to have a completely separate page in MODX' menu with save triggering an AJAX upload to the server. I wrote a PHP script to accept it on server side, which was about 40 lines of PHP code in its alpha stage before authentication and authorization. However when trying to use authentication and authorization functionality of MODX, I quickly discovered that I had to implement large amount of support classes sprinkled over several files. This became problematic as I could not determine which class is responsible for what, and the whole mass of boilerplate code quickly became unmanageable. This indicates bad design from my point of view, since it again defines policy one has to go through instead of just mechanism as could be expected and it also generates large amounts of confusing code without adding any real value. In the end, I have rather just exposed `svgGeo` as static HTML, JS and CSS files on a predefined URL and then added a menu item to MODX opening it in a new tab, effectively making it standalone and requiring the user to save the converted map to disk and upload it manually. Again, I must advise on moving away from MODX if another version of CTU Navigator was to be developed since the CMS, while convenient for the user, is a nightmare for a developer.

Figure 3.2: svgGeo applet appearance

Appearance of the conversion applet is shown in figure 3.2.

3.4 Push notifications

I was also tasked with implementing support for push notifications. Push notification is a method of notifying mobile platforms and other devices of

any kind of events happening on a remote machine. Since it is very hard to maintain permanent Internet connection to these devices, which would also have the downside of draining their battery, the simulated push-based strategy made available by push notifications is quite convenient. The notification process usually works by obtaining access key for one of the push notification providers for these devices and pushing the events onto the provider servers, which are then periodically checked by the devices. There were two platforms to implement push notifications for, the first is the Android platform and the other is the Windows Phone platform.

There are various providers that happen to aggregate multiple push notification targets under one common API, however since these have various conditions related to their uses and since the underlying mechanisms usually boil down to Firebase Cloud Messaging (originally Google Cloud Messaging) and Microsoft Push Notification Service, I have decided to use these directly.

3.4.1 Push notification algorithm

Both of the two push notification frameworks work similarly. The sending side, hereinafter called the Server, can send arbitrary messages, both textual and binary, to the receiver (called the Client from now on) by passing them to servers handling the final delivery (called the Provider). The algorithm is as follows:

1. The Client requests its identification token from the Provider and provides it to the Server.

2. The Server stores this token and reuses it for all later messages.

3. Once the Server needs to send a message, it sends it to the Provider along with the device token.

4. The Client periodically checks the server for any new messages and retrieves them (this is transparent to the Client's applications).

5. The Client receives and processes the message.

The steps 1 and 2 are usually performed once, while the rest are performed for every message sent.

I have implemented one new API call, `/store-token`, to store the devices identification tokens and a new MODX Manager menu item to send a message notifying the devices of a update. This support has its problems, though, as the Firebase Cloud Messaging requires implementation of exponential backoff and resending, which is impossible to do correctly just in PHP and would need operating system support.

3.5 Testing

3.5.1 User testing

Testing of the results uncovered several problems mainly in implementation of the backend parts. User testing of the two applets, svgGeo and wayEdit required testers with fair amount of technical knowledge because of the nature of the applets. This requirement was satisfied by a team of three testers working in the IT sector. They were given three testing scenarios. One testing

SVG to GeoJSON map conversion

1. Download one of the SVG maps from given address.
2. Open svgGeo conversion application.
3. Load the downloaded SVG map.
4. Set the locations of three shown reference points.
5. Find out and set GPS coordinates to the points.
6. Save the converted map.
7. Verify the result in online application available at `http://geojson.io`.

Figure 3.3: svgGeo testing scenario

scenario focused on svgGeo with a simple set of tasks shown in figure 3.3. Two other testing scenarios tested usage of wayEdit. First of them dealt with

Editing the navigation graph

1. Load one of available maps via the selection box.
2. Try adding, updating and deleting of waypoints and routes.
3. Save the map.
4. Verify that the navigation graph was saved via box below the editor.

Figure 3.4: wayEdit testing scenario

standard create, update and delete operations on waypoints and routes and is shown in figure 3.4. Second scenario had just two tasks and tested automatic navigation graph extraction from OSM metadata. During testing, it became

apparent that new users expect standard desktop GUI and corresponding type of controls despite the browser nature of the applets, to the point that their first action for solving most of the testing scenarios was a right click into the map area expecting a menu of actions, which was not implemented as I did not deem it necessary at first. On top of that it would also be challenging to implement it due to lack of support by the applets' underlying library Leaflet, therefore I have to leave this unchanged due to lack of time. Also, several of the testers requested that the svgGeo's reference points display a visual indicator showing that their geographical coordinates have been altered, which was easy to add. Third notable issue concerned integration of the Leaflet library. The JavaScript events, which were triggered for the applet detail dialog, seeped through to this library and caused problems when editing values. This was also easy to fix by merely disabling event propagation. I realized during testing that a descriptive manual has to be provided to use these applets. I had written it and included in the project documentation.

3.5.2 Unit testing

The rest of the backend, mainly the API part, was tested by a mock client written in Go programming language. The Go language is designed with a

```
package ctunavi_backend_tests

import (
    "testing"
    util "koderja2/testutils"
)

const (
    BackendURL = "http://localhost:4000"
)

func TestInfo(t *testing.T) {
    body := util.GetAndDecodeJson(t, BackendURL + "/info")
    util.Has(t, body, "default_language", "string")
    util.Has(t, body, "language", "array")
    util.Has(t, body, "language.0.key", "string")
    util.Has(t, body, "language.0.title", "string")
    util.Has(t, body, "language.0.icon", "string")
}
```

Figure 3.5: One of API functional tests written in Go

variety of packages for development of both server and client web applications.

It also comes with a testing framework specifically designed to write unit tests and benchmarks. This testing framework is primarily targeted for integration and testing of projects developed in Go, yet it is quite versatile and easily used to perform functional tests on completely separate web applications. Due to their nature, MODX Snippets are fairly badly testable via unit tests, white box testing or any other type of tests designed for testing of isolated parts of code. This makes functional testing a necessity. I have extensive experience with Go testing framework therefore I have chosen it for this task as well. I used it to prepare several tests, each querying and testing output of one of the API entry points. These tests are heavily dependent on preparations made beforehand on tested MODX instance as I have not found any easy way to automate and repeat those preparations. Excerpt from one such test can be seen in figure 3.5.

3.6 Documentation

I have created several support texts to accompany the backend to make it easier to use. I could not use any standard PHP documentation tool as the code is inaccessible. This is due to the code being stored in database and, as far as I know, the only standard access route to is MODX' Manager. This made me provide all of the documentation manually in PDF format. The documentation includes API documentation for all of the API entry points and a user manual for the two applets as well as instructions for installation of my extensions to MODX. Since a lot of the code is contained in MODX Snippets, stored in the database and I am not aware of any documented way to export and import it, the easiest way to install the results of my work may be summed into simple "Clone the project's MODX instance database and files, migrate these to a new machine and tweak them to suit their new use".

Conclusion

In this thesis I have devised and implemented methods, processes and applications to transform images into vector-based maps, to create navigation graph on top of them and supply them in combination with other text and image-based content to several client applications developed under the CTU Navigator project. I have designed an algorithm based on linear algebra methods used to project image map points and lines onto the latitude and longitude coordinate system. The work done within this thesis forms a basis required for making the CTU Navigator project work on the server side; during my cooperation with the rest of the team we determined several ways to further expand this project. One such way would be to use QR codes or other types of image encodings to partially replace outdoor localization technologies, such as GPS, and to allow users to find out where they currently are. Another way would be to use other methods being devised by other students in parallel to this work, such as localization by WiFi. Apart from these, it would be possible to augment the navigation data by incorporating current blockades such as renovations, area leases, elevator loads and other into the graph data. Another types of content data may also be supplied to the backend clients, allowing them to cover a lot more information about the university instead of just pathing and events. The project is still in its infant stages and its possibilities are enormous.

Bibliography

[1] National Research Council. *The Global Positioning System: A Shared National Asset*. Washington, D.C.: National Academies Press, 1995, ISBN 978-0-309-05283-2.

[2] Allan, D. W.; Weiss, M. A. Accurate Time and Frequency Transfer During Common-View of a GPS Satellite. May 1980, [Online; Accessed 23 November 2016]. Available from: `http://tf.nist.gov/time/commonviewgps.htm`

[3] Anisetty, P. R.; Huang, C. Y.; et al. Using GPS to improve tropical cyclone forecasts. August 2014, [Online; Posted 24 August 2014]. Available from: `http://www2.ucar.edu/atmosnews/just-published/12183/using-gps-improve-tropical-cyclone-forecasts`

[4] Geology Department, Central Washington University. Real-Time GPS Data Analysis. [Online; Accessed 23 November 2016]. Available from: `http://www.geodesy.cwu.edu/realtime/`

[5] Čermák, O. *ČVUT Navigátor*. Phd dissertation, Czech Technical University in Prague, February 2012.

[6] Google Inc. Google Maps API. November 2016, [Online; Accessed 23 November 2016]. Available from: `https://developers.google.com/maps/`

[7] Mikeš, S. *ČVUT Navigátor III - řízení vývoje multiplatformních aplikací*. Bachelor's thesis, Czech Technical University in Prague, May 2016.

[8] Horvat, J. *ČVUT Navigátor III - návrh a implementace multiplatformního klienta*. Bachelor's thesis, Czech Technical University in Prague, May 2016.

[9] MODX LLC. MODX CMS. 2004–2016. Available from: https://modx.com

[10] Levin, R.; Cohen, R.; et al. Policy/mechanism separation in Hydra. *SOSP '75 Proceedings of the fifth ACM symposium on Operating systems principles*, November 1975, online; Retrieved 9 December 2016. Available from: http://dl.acm.org/citation.cfm?id=806531&dl=ACM&coll=

[11] Sobel, I. History and Definition of the so-called "Sobel Operator", more appropriately named Sobel-Feldman Operator. February 2014. Available from: https://www.researchgate.net/publication/239398674_An_Isotropic_3_3_Image_Gradient_Operator

[12] Jankovics, A. ImageTracer. 2015–2016. Available from: https://github.com/jankovicsandras

[13] Selinger, P. Potrace. 2001–2016. Available from: http://potrace.sourceforge.net

[14] World Wide Web Consortium. Scalable Vector Graphics (SVG) 1.1 (Second Edition). August 2011. Available from: https://www.w3.org/TR/SVG

[15] Butler, H.; Daly, M.; et al. The GeoJSON Format. Internet Requests for Comments, August 2016.

[16] Ivison, R. *Bézier Curves*. Honors thesis, Southern Connecticut State University, May 2011. Available from: http://webcache.googleusercontent.com/search?q=cache:PZ7GonDR5MsJ:www.southernct.edu/mathematics/uploads/textWidget/wysiwyg/documents/Rachael_Ivison_2011.pdf+&cd=2&hl=en&ct=clnk&gl=us

[17] GNU General Public License. June 1991. Available from: https://www.gnu.org/licenses/gpl-2.0.html

[18] GNU Lesser General Public License. June 2007. Available from: https://www.gnu.org/licenses/lgpl-3.0.html

[19] FreeBSD License. Available from: https://github.com/Leaflet/Leaflet/blob/master/LICENSE

[20] Open Knowledge International. Open Database License v1.0. September 2012. Available from: http://opendatacommons.org/licenses/odbl/1.0

[21] Myška, M.; Polčák, R.; et al. Veřejné licence v České Republice, version 2.0. [Online; Accessed 26.11.2016]. Available from: http://is.muni.cz/repo/1203341/Myska_et_al._-_Verejne_licence_2.0_-_online.pdf

[22] Consolidated text of Act No. 121/2000 on Copyright and Rights Related to Copyright and on Amendment to Certain Acts (the Copyright Act), as amended by Act No. 81/2005, Act No. 61/2006 and Act No. 216/2006. 2000–2006. Available from: https://www.mkcr.cz/doc/cms_library/12-az_2006_v_aj-2005.pdf

Appendix A

Glossary

API Application Programming Interface, an interface for two programs to communicate over

app Short for application, mostly used with mobile phones

applet Small utility application, usually with single function

BSD Berkeley Software Distribution, a Unix operating system derivative developed in Berkeley

CMS Content Management System, an application used to manage digital content

CSS Cascading Style Sheets, a language used to describe presentation for other markup languages

FURPS Functionality Usability Reliability Performance Supportability

GNU "GNU's Not Unix !", an extensive collection of free software forming an operating system

GUI Graphical User Interface, a type of user interface

HTML HyperText Markup Language, a language used to create web pages

HTTP HyperText Transfer Protocol, a type of protocol originally used to transfer web pages

IT Information Technology, a field using computers to manipulate information

JSON JavaScript Object Notation, a data formatting language

OS Operating System, base program running on a computer

QR code	Quick Response code, a type of two-dimensional barcode
SVG	Scalable Vector Graphics, a vector image format
XML	Extensible Markup Language, a data formatting language
WiFi	Wireless Fidelity, a wireless communication standard

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `docs` .... directory with documentation for the practical part of the thesis
- `src` ..... directory containing all source code
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format