Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Temporal Map of Switzerland

Semester Thesis

Florian Zinggeler

zifloria@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Manuel Eichelberger
Prof. Dr. Roger Wattenhofer

June 13, 2016

Abstract

In this project, I created an interactive map of Switzerland to bring the past closer to the present. The map displays old and new aerial photographs and allows the user to easily compare the same location at different times.
In this paper, I will present how the images were obtained and how they were automatically georeferenced and warped such that they fit nicely on top of each other.
A fully automatic solution is presented that performs all needed steps automatically. This method works well with pictures taken above a certain height. For images where the system fails, a simple manual process can be used to fill the gaps. Both approaches combined could be used to eventually cover the whole of Switzerland.

Keywords: Georeferencing, computer vision, aerial photos, GIS, image registration

Contents

Chapter 1

Introduction

Recently, the Federal Office of Topography, Swisstopo, released a huge part of its national treasure[1]: over 200000 aerial pictures of Switzerland, dated from 1926 to 2007. They are publicly accessible on their LUBIS viewer website [28]. On this map, we can see when and where the pictures were taken. By clicking on an image we can view it in a separate window. While this way of viewing is certainly useful at times, the user cannot really see the bigger picture this way. Since we can only view one image at a time, it is hard to compare the same location at different times. It is even harder to get an overview of the whole area if the images were not taken high enough, as is the case with many of the older images.

Wouldn't it be nice if we could see the whole of Switzerland just as with the normal aerial view, but at much earlier dates? This question was the main motivation for this project. The idea was to use the images made available by Swisstopo, merge them onto a single giant image per date and display the result in an interactive web application created specifically for the purpose of presenting the past.

1.1 Goals

The goal of this project was to create an interactive map of the past.
The task could therefore be divided into two main tasks: Creating the user interface and processing the images into an appropriate format for the frontend. Accordingly, this report is divided into two main chapters. In the first chapter, the methods used to obtain the images and to process them such that they can be later used by the frontend, are presented. The second part describes how the web based map application was created. Then, there is an additional chapter that gives an outlook to the future, where several ideas are presented that could

[1] Historic Geodata, Key data and features, http://www.swisstopo.admin.ch/internet/swisstopo/de/home/topics/geodata/historic_geodata/im_coll/key_dat.html, Accessed: June 13, 2016[21]

be implemented for better results.

To bring the images into an appropriate format, they first had to be obtained and heavily processed. The difficulties here were that each image was taken at a different time, with a different camera, from a different height and not exactly straight down. Additionally, the older images had a much lower resolution than the newer ones. To make matters more complicated, all images had a frame around them, but not always the same one. This made an automated solution to remove this frame slightly more complicated.

The map application should allow the user to easily choose between different times. The idea was to provide different controls, such as a slider or a dropdown. Also, it should be easy to compare the same location at different times. For this, a way to split the view into two parts seemed to provide the best solution.

1.2 Related Work

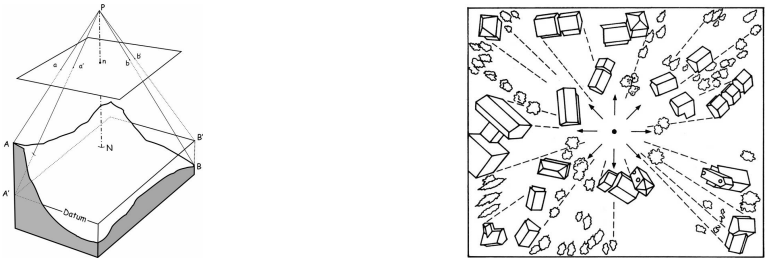
This is not the first time something similar was created. The processes of georeferencing and orthorectification are well understood, as well as displaying tile maps on the web.
This project is a follow up to an almost identically named project by Marc Müller [38] in Spring 2015. However, he took a different approach in both the design and functionality of the web application as well as in the method used for processing the images. He used a method called image stitching, as presented in the paper by Brown et al. [5] to merge the pictures together. This is mainly used in creating panorama photos and works very well for small areas, however errors accumulate with area size and this method is therefore not very useful for my application here.

For automatic georeferencing, there exist various approaches. Some use an area based method based on cross-correlation or mutual information such as in [6] by Bunting et al. Others try to find matching features between a reference image and the old images via edge detection, as in [29], [16] or [52]. Another approach is to vectorize the data. For this, roads and other line features need to be detected. In retrospect, this approach seems to be most promising, as shown in [32], [33], [55] and [7].

Chapter 2

Automatic Georeferencing

Georeferencing is the process of assigning every pixel on a picture to the correct geographic location. Usually, this requires some control points on both the picture and on an already accurate map of the same physical location as a reference. At least three points are required, but the more ground control points are used, the more accurate can the pixels in between be assigned to their location. More than three points are required because the points have to not only correct the location, but also the lens distortion of the camera, the uneven terrain and the perspective distortion. (See Figure 2.1). In cities or flat areas we can mostly neglect the terrain deformities, however in more hilly areas or even the Alps this has to be accounted for. Currently, the assignment of these ground control points is mostly a manual process, supported by a GIS software such as the free and open-source QGIS[44] or the commercial ArcGIS [42] software. To my knowledge, there does not exist any free software that can automatically georeference an image. On the commercial side, ArcGIS is capable of automatically assigning ground control points, however even their solution "does not work well with scanned maps or historical data"[1].



(a) Terrain distortion

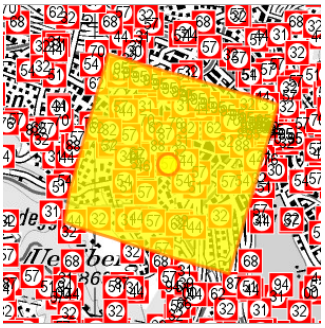
(b) Perspective distortion

Figure 2.1: Reasons that make georeferencing difficult. (From [22], [23])

[1] Georeferencing a raster automatically, http://desktop.arcgis.com/en/arcmap/10.3/manage-data/raster-and-images/georeferencing-a-raster-automatically.htm, Accessed: June 13, 2016[17]

2.1 The Raw Data

The data used for this project were the aerial images available at LUBIS [28]. These pictures have some metadata associated with them, as seen in Figure 2.2. Of most interest to us is the approximate location, the approximate rotation and the approximate flight height. Other useful information are the pixel dimensions of the pictures and the date the pictures were taken. This information can be obtained by either clicking on the 'more info' button in the LUBIS viewer[2] or, more useful for programmatic access, by using their JSON REST API [53]. With this information we can get a good first guess for the actual geolocation.



Aerial Images swisstopo b / w (Federal Office of Topography swisstopo)

Picture number	19579990082634
Inventory number	194033
Flight date	28-06-1957
Flying height	4700
Scale	1:20200
Centre coordinate Y	681400
Centre coordinate X	247328
Film type	bw
Dimensions of original image	18 x 18 cm
File size	83
Image path	lubis\frame\000-194\000-194-033.tif
Ortho image path	-
Orientation elements	-
Rotation	16

Figure 2.2: An image with its metadata. (From [51])

The pictures themselves come in various shapes and sizes. They have different aspect ratios, different resolutions and different colours (not just grayscale vs. colour, but also the sepia tone that old pictures tend to have). Additionally, they depict different sized regions of the map, some of them with a very small viewport, just showing a close up of the city while others show the whole of Zürich. On top of that, each picture is surrounded by a dark frame. Additional metadata is displayed on the frame, such as the exact time the picture was taken plus some information about the camera. Older images have measurement devices lying on the frame, such as a watch or a barometer. An additional hurdle is present when obtaining these images. Although they are publicly accessible to view, they are not intended for downloading. Since the images are divided into many subimages, downloading them automatically was slightly complicated.

[2] Example metadata, https://api3.geo.admin.ch/rest/services/swisstopo/MapServer/ch.swisstopo.lubis-luftbilder_schwarzweiss/19320480121010/extendedHtmlPopup?lang=en, Accessed: June 13, 2016, [9]

2.2 Obtaining the Data and Preprocessing

2.2.1 Getting the Images

The LUBIS online viewer displays these images using a popular JavaScript library called OpenLayers [40], by using a Web Map Tile Server (WMTS [36]). A WMTS server is organised in a straightforward way. The URL plus the file name describe the coordinates and the zoom level of the tile. By knowing this, it was possible to download one image by downloading all tiles on the highest zoom level. After that, I used the popular image processing tool imagemagick [27] to merge them together to get a single high resolution image. As a penalty for doing it this way, the downloaded images have a barely visible watermark on them.

2.2.2 Rough Georeferencing

To position our images correctly on a map, we need more information about the camera. We need to know the focal length to guess the scale of our images. However this information is not available in the metadata, and we have to estimate it. Luckily, the bounding box of our images is provided in the metadata. The bounding box specifies the lower left coordinate and the upper right one. Together with the rotation, we can calculate the actual image dimensions. (See Figure 2.3).

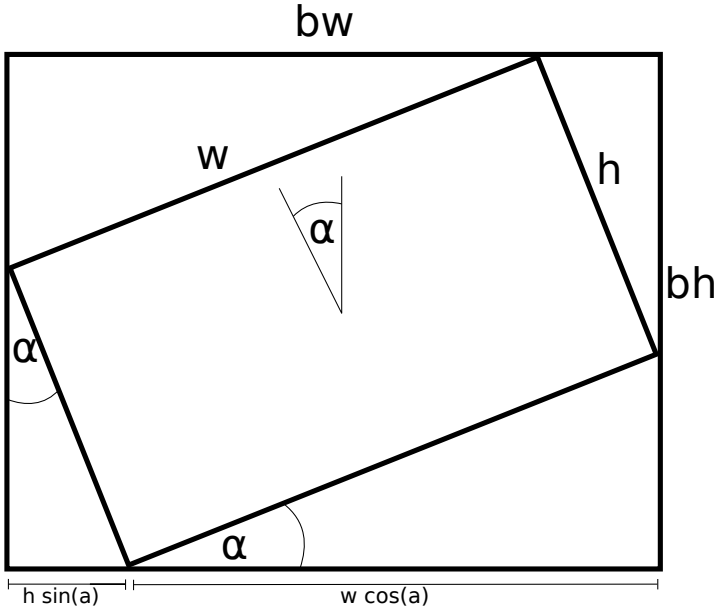
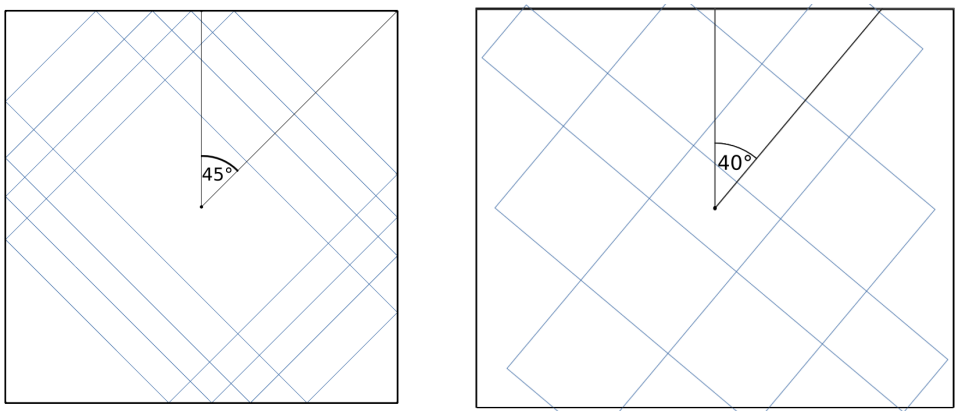


Figure 2.3: The bounding box of an image. We need to find out w and h

However, if we only use this information, we end up with the following equation which is numerically unstable and completely wrong when our rotation gets close to 45°.

$$w = \frac{-bh \cdot |\sin\alpha| - bw \cdot |\cos\alpha|}{|\cos\alpha|^2 - |\sin\alpha|^2} \qquad h = \frac{-bh \cdot |\cos\alpha| - bw \cdot |\sin\alpha|}{|\cos\alpha|^2 - |\sin\alpha|^2}$$

This would not be such a problem if the given bounding box was perfectly accurate, as we could add a special rule for this case. However, since the bounding box is not exact, we get even worse results using this simple solution. The instabilities arise from the fact that given a square as the bounding box and an angle of 45° we get infinite solutions, as illustrated in the Figure 2.4 below. For other inputs, no solution exists. In our case this would occur quite often, as the bounding box is not very accurate.



(a) Infinite valid solutions at rotation 45°

(b) Given these parameters, no exact solution is possible. The result found by the naïve solution is so large that it lies far outside this paper. The depicted solutions are from the other approach, given different aspect ratios r.

Figure 2.4: Reasons that make the naïve solution unstable with errors in the given parameters.

To get a more robust solution for all angles, we can make use of an additional parameter, the pixel dimensions. Thanks to this, we know the ratio of the picture width and height $r = \frac{pw}{ph}$ and get a much more robust equation. This is also stable with small errors in the bounding box. We get the modified equations by simply substituting w by $h \cdot r$ or h by $\frac{w}{r}$. This way we get two solutions for w and h so we just take the average of both solutions.

$$h_1 = \frac{bw}{|\sin\alpha| + r|\cos\alpha|} \qquad w_1 = \frac{bw}{\frac{1}{r}|\sin\alpha| + |\cos\alpha|}$$
$$h_2 = \frac{bh}{|\cos\alpha| + r|\sin\alpha|} \qquad w_2 = \frac{bh}{\frac{1}{r}|\cos\alpha| + |\sin\alpha|}$$
$$h = \frac{h_1 + h_2}{2} \qquad w = \frac{w_1 + w_2}{2}$$

Together with this information and the position and rotation from the metadata, we can create a first rough georeferenced image. I used the GeoTIFF format [46] for this, as it is capable of storing all the needed metadata alongside the image in the same file. A GeoTIFF is simply a TIFF file, with a very specific set of attributes that need to be present. This had the advantage that I could use existing GIS software such as QGIS [44], GrassGIS [20] and uDig [54] to visualize and interact with the data early on. The GeoTIFF format has an attribute that specifies the transformation from pixel space to some geographical coordinate system. This transformation is specified by an affine transformation, that is a matrix containing the rotation, scale and position information. Figuring out this transformation was harder than expected, as all references I found only made use of the position and scale, but not any rotation. The fact that our images are rotated around the centre and not the origin of the picture made it even more challenging, as this affects the position. Nevertheless, after some experimentation, I managed to derive the correct transformation:

$$\hat{x} = \begin{bmatrix} \frac{w}{p_w}\cos\alpha & -\frac{w}{p_w}\sin\alpha \\ -\frac{h}{p_h}\sin\alpha & -\frac{h}{p_h}\cos\alpha \end{bmatrix} x + \begin{bmatrix} c_x + \frac{-w\cos\alpha + h\sin\alpha}{2} \\ c_y - \frac{-w\sin\alpha - h\cos\alpha}{2} \end{bmatrix}$$

x are the pixel coordinates, $\hat{x}$ the geo coordinates, p_w, p_h the width and height of the image and c_x, c_y the centre point of our image in geo coordinates

2.3 Removing the Frame

To seamlessly tile our photographs, we first have to get rid of the frame surrounding each picture. Here are two examples of how this frame may look:

Figure 2.5: Two examples of the black frame. (Images from [1])

To move along quickly and not slow down the development of other parts of this project, I initially cropped these frames manually. One small hurdle here was that our images contain additional geodata, which gets lost if the images are processed with a normal image editing application such as GIMP [18]. To overcome this, I used GIMP only to read out the pixel coordinates of a selection and used this information to do the actual processing with gdal_translate [14]. This way the new metadata is produced correctly.

Later, this process was automated using the powerful open-source computer vision library OpenCV [39]. Our task is to find the biggest possible rectangle that does not contain any parts of the border. To do this automatically, we need to define what it takes for a pixel to be part of the frame. What worked nicely was the following formula. It classifies whether a line of pixels is part of the frame or not.

```
def is_frame_line(pixels):
    num_frame_pixels = 0
    for pixel in pixels:
        if pixel <= min_black or 255 <= pixel:
            num_frame_pixels += 1
    return num_frame_pixels/len(pixels) > threshold
```

Where a pixel is described as its brightness, from 0 to 255. We chose the parameters *min_black* and *threshold* to be approximately 20 and 0.01 respectively. You might wonder why the threshold was chosen as such a low percentage. The reason is that some of the pictures have round objects on the frame that are part

of the frame, so it should detect even just a small part of those objects. Besides, the images normally do not contain many near black pixel values, so even with a threshold such low, there were no false positives in practice.

The algorithm for finding the biggest possible rectangle to crop the image works similar to a binary search. There is an inner rectangle that is definitely not on the frame, and there is an outer rectangle that is definitely on the frame. Then, in each step, the rectangle in the middle is computed. Then, each side of the rectangle will be checked using the formula defined above to find out whether it is intersecting the frame or not. If it is, the side corresponding to the line that was checked on the outer rectangle will be set to the middle one. Conversely, if it is not part of the frame, the inner rectangle's side will be adjusted. This algorithm converges very quickly; with our images we rarely needed more than ten iterations. The full algorithm can be found in Appendix A.

2.4 Georeferencing

2.4.1 Area Based Image Registration for Georeferencing

Image registration is the process of aligning two or more images of the same scene into a unified coordinate system. It is often used in medical imaging and also in remote sensing, among others.

In a first approach, I tried to use an automated image registration algorithm provided by The Remote Sensing and GIS software library (RSGISlib) [49]. The library provides various useful tools for remote sensing, such as image calibration or an image registration module, just what I needed. This module implements an area based algorithm for image registration, as presented in the paper by Bunting et al.[6]. Unfortunately, I got very unsatisfactory results using this method. The algorithm found way too many false positives, making an accurate transformation impossible. Additionally, even with a very low resolution image it took hours to find enough matches. The algorithm has an adjustable search window that specifies how many pixels around some initial tie points are searched. When this value was set too low, it did not find any points at all. If it was set too big, it would not finish within any reasonable time.

I concluded that an area based method is not suitable for our data, as others have before me: "feature-based methods [are] recommended if the images contain enough distinctive and easily detectable objects. This is usually the case of applications in remote sensing and computer vision. The typical images contain a lot of details (towns, rivers, roads, forests, room facilities, etc). On the other hand, medical images are not so rich in such details and thus area-based methods are usually employed here." [56]

I therefore was looking for another solution. To start working on the frontend

I needed some nice data, however the tried method would not give me any satisfying results. Since the available time for this project was limited, I had to move on.

2.4.2 Manual Georeferencing

Since my first try at automatic georeferencing failed, I started doing the georeferencing process manually, using uDig [54] and QGIS [44]. This way, I could move on to the next step even though the first step was not satisfying yet. By placing a moderate amount of ground control points in a GIS software, I was able to get much better results than with the area based attempt before, at the cost of doing each picture manually. To get satisfying results, I had to place approximately 16 evenly distributed points per image.

Using this method, I could use the obtained ground control points to warp the image such that it fits nicely on top of the reference map. To do this, I used the very versatile GDAL [11] application, namely its gdalwarp command [15]. This utility is an image mosaicking, reprojection and warping tool that is also able to warp an image according to some transformation specified by ground control points. I compared different interpolation methods and decided that tps (thin plate spline)[4] gave the best results.

I used this process to create a sample set of georeferenced images that could then be used for the last part of the image enrichment process, the creation of tile images.

2.4.3 Georeferencing Using Feature Matching

After experimenting with the manual georeferencing process, I took a second stab at automatic georeferencing. This time, equipped with some experience with openCV from the frame cropping, I had a better idea of how such an algorithm might work. The algorithm is supposed to match features of the image to be registered onto the base image. Therefore, the algorithm should produce a mapping from pixel coordinates of one image to the coordinates of the other image. In our case, we are given an image and its approximate bounding box (floating image) and the corresponding part of the base image, which is already georeferenced. By matching features from one image to the other, we can figure out the correct coordinates for our floating image.

The process I ended up using can be divided in these steps:

1. preprocessing
2. feature detection and description
3. feature matching
4. filtering
5. postprocessing

The individual steps are explained in the following paragraphs.

Preprocessing

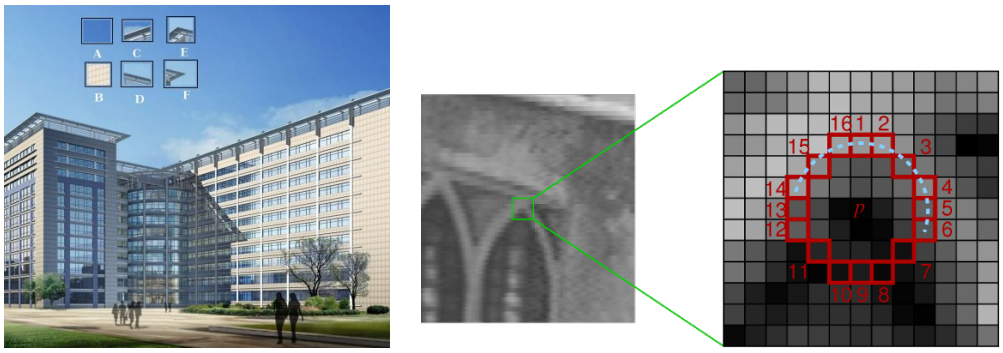
In a first step, a base image from the same bounding box as the floating image is obtained. The higher resolution image is then scaled down, such that one dimension matches the dimension of the other one. This rescaling is not strictly necessary for the algorithm to work, but experiments showed that it performed better when both images had similar dimensions. Afterwards, both images are scaled down, mainly to reduce the processing time. This rescaling barely affected the results, as long as the resolution was not lowered below a certain level. A good trade-off appeared to be at around 1000 by 1000 pixels, as at that resolution, we still got nearly identical results as with the full resolution, but at a much lower processing time. Even though a rotation invariant feature matching algorithm was used, the base image was rotated such that both images were oriented roughly the same way. This made one of the later filtering steps easier.

Both images were converted to a grayscale image, as the used algorithm for feature matching only works with grayscale images. Different image filters were tried hoping they would improve the result. Histogram equalisation[41] did improve the results significantly, so this filter was applied to both images.

Feature Detection and Description

In the next step, we had to identify potential features in both images and compute a description for them such that they could be compared later.
To identify good features, we had to find places in the image that were unique enough, such that we could easily find the same place in another image. Examples of good features are street corners, buildings, bridges, etc. These are generally referred to as corners. Contrast these to some point on a lake or the middle of a crop field; these points are incredibly hard to retrieve in another image, even for humans, as illustrated in the figure below.

The descriptors work by computing a vector based on the surroundings of a feature. An important property of a descriptor algorithm is whether they are



(a) In this example, good features would be E and F. For the others, it is very hard to find where they are from.

(b) FAST corner detector

Figure 2.6: A visual description of good features. (From [24], [25])

scale and/or rotation invariant.

There are various algorithms that can be used for both feature detection and feature description. The most popular ones are SIFT[34], SURF[3], ORB[50] and BRISK[31]. All of these come with both a feature detector as well as a feature descriptor. However, there are other approaches such as FAST [47][48] that can only be used to detect features, not to compute descriptors. It is still useful, as a description for the features found by FAST can be computed later with some of the above mentioned algorithms. In fact, ORB uses FAST to find features.

I chose ORB for both feature detection and feature description. The reason for choosing ORB over SIFT or SURF was that SIFT and SURF are patented, while ORB is not. I gave all above mentioned algorithms a try to compare their performance on our task. With our pictures, they all performed about equal. When ORB performed bad, all the others did too and when ORB performed good, all the others did as well. The only big difference was the execution time, where ORB and SURF were clear winners, as shown in table 2.1.

Method	Run time (seconds)	Features	Good matches
ORB	6.636647	10000	15
ORB	22.574342	20000	17
SURF	50.757283	19026	10
SIFT	27.395539	10000	12
SIFT	97.436219	21315	20
BRISK	601.118143	82195	54

Table 2.1: Comparison of execution times and number of matches for different feature detection algorithms. **Note**: This is no fair comparison, as different amounts of features are detected. The number of good matches can serve as an indicator of the quality of an algorithm.

Feature Matching

Features from one image can then be compared to those of the other one. To get a measurement of the similarity between two features, we computed the distance of their descriptor vectors.

By simply taking the best matches, we unfortunately get many false results. To better filter the matches later, we need to compute for each match both the best and the second best match. This additional information about the second best match is needed in the next step to filter out the more unique features, that is the ones where we are pretty sure they are correct.

Filtering

After using a brute-force matcher to match all the features, we need to filter out the worst matches. To this end, I used three filtering stages. The order of the first two methods does not matter, but the third step has to come last.

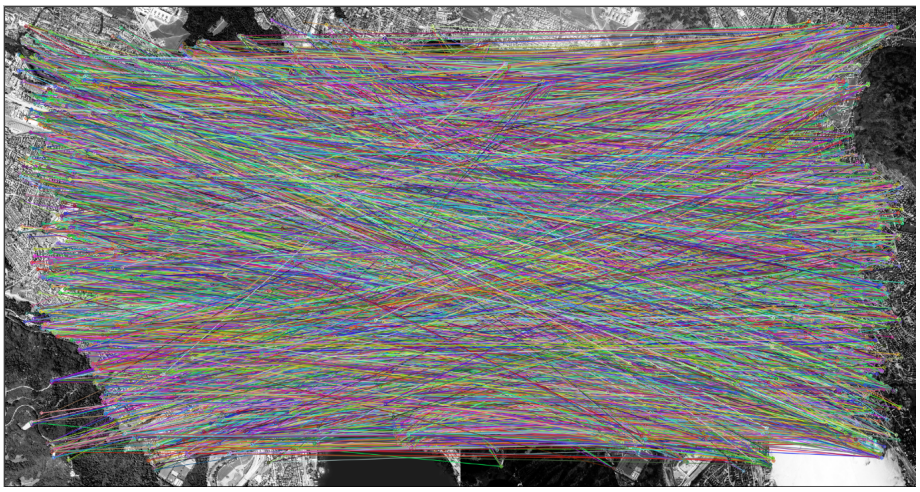


Figure 2.7: All matches, what a mess! (Images from [1])

In the first step, a ratio test of the best and second best match is performed to filter out all matches that are not unique enough to be accurately matched. For further information on why this works well, I recommend the paper by Lowe[34], where this method was proposed.

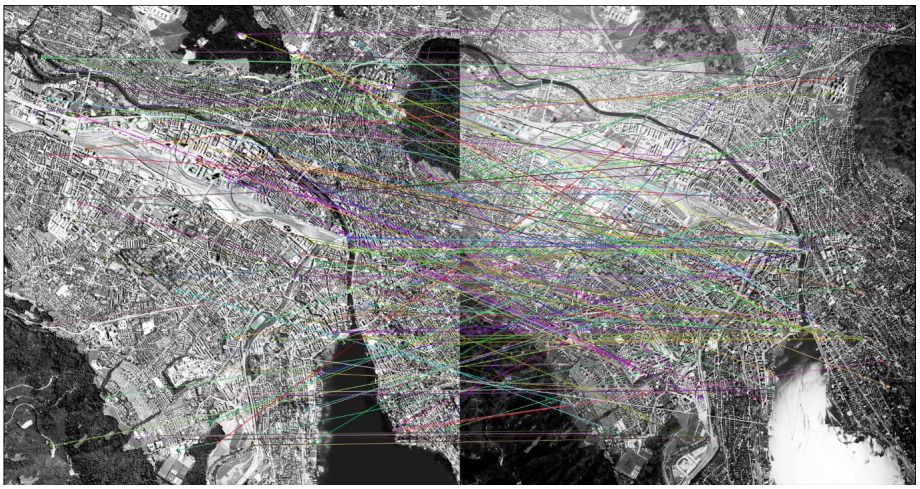


Figure 2.8: Filtered matches after performing the ratio test. We are only left with a few hundreds of potentially good matches. (Images from [1])

As a second step, we can discard all matches that could never ever be valid, given our initial guess at the position, rotation, scale, flying height and focal length. Using these quantities we can make a prediction where a point should lie. Then we can simply use the distance from the predicted point to the matched one and filter out all points that are too far off.

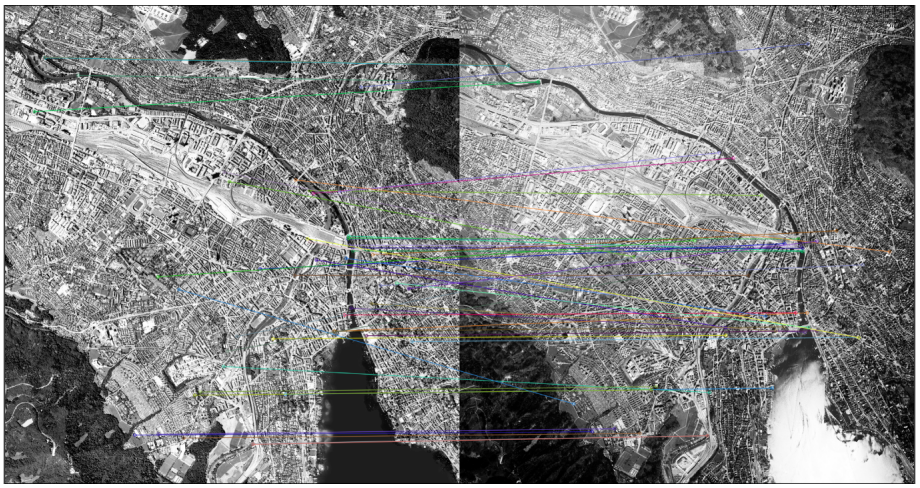


Figure 2.9: Filtering out all matches that are too far off the predicted point. (Images from [1])

Now, we are still left with many wrong matches plus a few correct ones, as can be seen in Figure 2.9. To get only the correct ones, we need to fit our matches to some model, and filter out all the outliers. This model is a description of a perspective projection. To filter out the outliers, I used the RANSAC algorithm [10] which is an iterative method to estimate the parameters of a model from a set of data points containing many outliers. Therefore, it can also be used as an inlier detector. This leaves us with a small set of matches, which approximately form a perspective projection and are hopefully correct. An example of the result is shown in Figure 2.10.

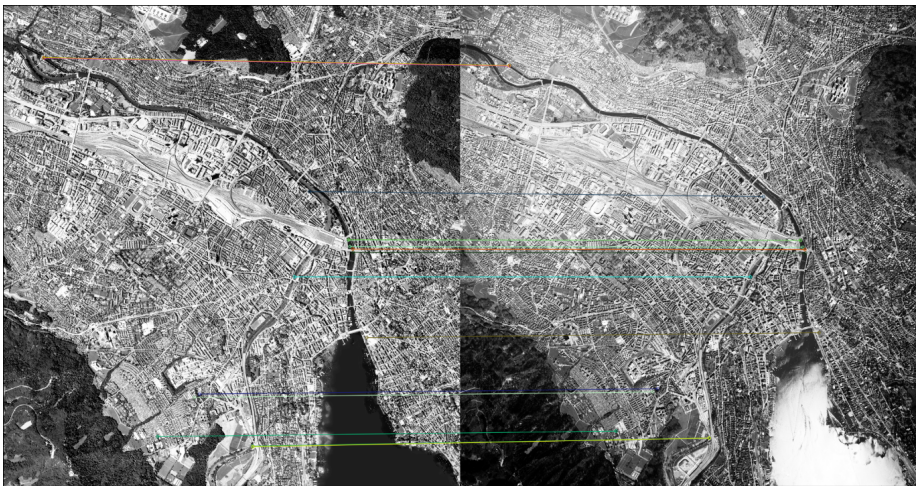


Figure 2.10: Only correct matches are left after performing RANSAC. (Images from [1])

Postprocessing

Now that we have our probably correct matches, we need to get the initial coordinates back since we skewed them in the preprocessing phase by scaling and rotating. Luckily, this a simple matter of performing the transformations in reverse. With our initial coordinates back, we can finally georeference our image. With the correctly georeferenced base image, we can assign ground control points to our floating image. Then, we can warp the image using the same method as with the manual georeferencing, described in section 2.4.2.

2.4.4 Tile Image Generation

To display maps in a web browser, we cannot simply display the whole image, as the resolution of these images is way too big for rendering, let alone for serving them. One single image has an uncompressed size of roughly 600MB (compressed with JPEG ca. 60MB), at a resolution of approximately 16000 by 16000 pixels. When all the images from one date are merged into a single layer, the merged picture is even larger.
For this reason, map applications for the web usually use tiled images to only download and render what is really needed. An additional benefit of doing this is caching. This way, the client does not have to download the same image over and over again and thus puts less load on the server.
These tiled images are pre-generated on the server, such that they cover the whole area of the picture at various zoom levels. This obviously trades access speed for disc space, as we now need all our images more than once at different resolutions.
The Open Geospatial Consortium has defined a standard for these Web Map Tile Servers (WMTS) [37], where it is clearly specified how these tiles are organized into zoom levels and coordinates. Following that specification, the whole world fits into a 256 by 256 pixel tile at zoom level zero.

To produce tiled images, I could again make use of the versatile GDAL tool set [11] for both merging multiple images into one and generating tiled images from the merged files.

To merge all images belonging to a specific layer (containing all images of a given date), I used the .vrt file format [12], specifically made for such applications. This format is a text based XML specification of the merged file. Thanks to the virtual file format, we can work with many images at the same time as if they were merged into a single image. The format also allows to define transformations and filters on the images, which will later be applied while processing. To merge all my files, I made use of its capabilities to map nodata[3] values to transparency

[3]These are special pixel values such as pure black, which are marked this way to specify that they do not contain any information

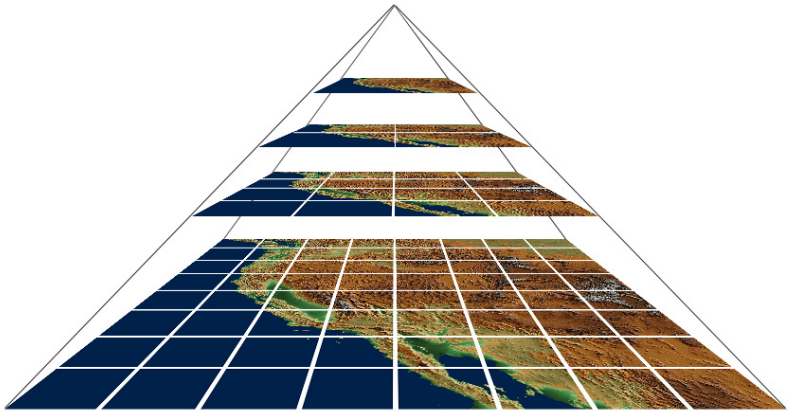


Figure 2.11: Tile pyramid. (From [26])

to merge the images without any black borders.

Finally, using gdal2tiles.py [13] I was able to render the tiles from the previously created merged .vrt file. gdal2tiles.py is even able to produce a simple skeleton for a web application to display the produced map. However, since my needs were more complex and I was not using any of the supported map libraries, I did not make use of this capability.

2.5 Putting It All Together

Every step of the pipeline has now been laid out. All that is left is some way to hold it all together. Throughout the whole project, I developed various Python [43] scripts for the individual parts of the whole process.
In the end, this evolved into a simple script that controls the whole process, from downloading the images to generating tiles.
This script needs as input one or more bounding boxes in the Swiss coordinate system plus optionally some image IDs directly taken from the Swisstopo LUBIS viewer [28]. Given that, it will download all the images that are available within these bounding boxes, join them together and perform a rough georeferencing using the available metadata.
Then, features will be searched and hopefully matched between the converted images and a base map. If the matching was successful[4], it will warp the image accordingly. In the end, the images of one date are merged into a single layer and new tiles will be generated for them. At the same time, a file containing metadata about all the available layers will be updated, since this file is needed by the frontend. In the merging phase, pictures that were manually georeferenced

[4]See section 4.2.2. Whether the matching was successful or not is based on the number of matches.

take precedence over the automatically georeferenced one. This way, we can correct mistakes or add some that failed completely. The script also provides some statistics about how many pictures were warped successfully.
The diagram below provides a good overview of the whole process.

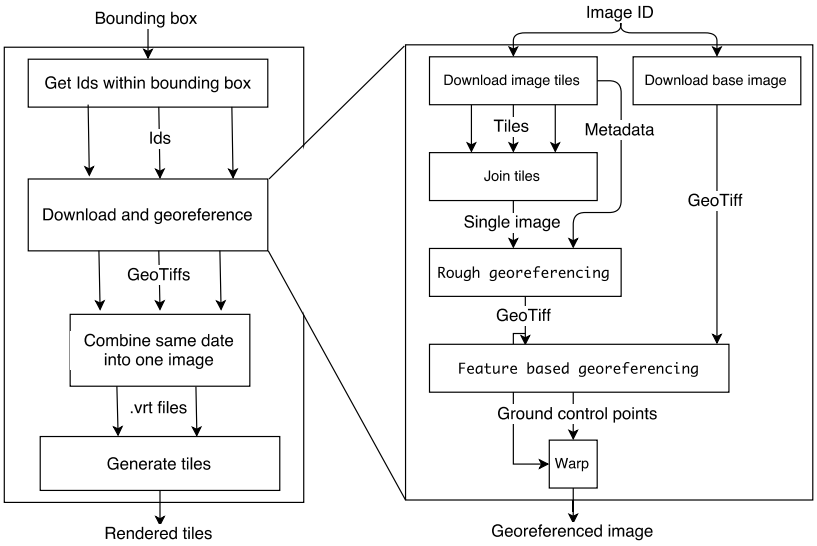


Figure 2.12: Diagram of the whole process

Chapter 3

The Frontend Application

To make our results easily accessible, we chose to create a web application to view the finished map of the past and present. To do this, we need some way to display maps in the browser and a nice interface to control the application.
To display our tile images, a number of map frameworks for the web were considered such as OpenLayers 3 [40], Mapbox [35], Leaflet [30] and Google Maps [19]. All of these could get the job done, so it was more a decision of taste and less a technical one. I ended up using OpenLayers, as their API was well documented, the framework widely used and it comes with a friendly license (BSD 2).

Since web developers have to invent everything a thousand times over, there exist a plethora of solutions for the same problem, namely complex user interfaces and user interaction. Because of that, various frameworks, libraries and even languages were considered such as React [45], AngularJS [2], Elm [8], etc. However, since only a small fraction of what these solutions provide would have been needed and since the requirements weren't too complex, I choose the plain old holy trinity of JavaScript, HTML and CSS to do the job, without any fancy UI library.

Each merged picture, that is all pictures captured on the same day, were put on a different layer. The oldest layers were put on top of the newer ones, up to the newest one, the current orthophotos provided by Swisstopo. This way, a date to be viewed can be selected by hiding all dates older than the one we are interested in.

To control the date, three different ways were implemented. One is a slider such that we can easily see the changes of two succeeding years. To complement that, the left and right arrow keys were mapped to go to the next and previous date. If we are looking for a specific date, a dropdown is provided for precise navigation. Because we wanted to be able to compare not just two successive

dates, but any with any, a button for splitting the view was implemented. This way, we can view the same location at two different times, side by side. These two viewports are then synchronised, such that they always show the same location at the same zoom level.

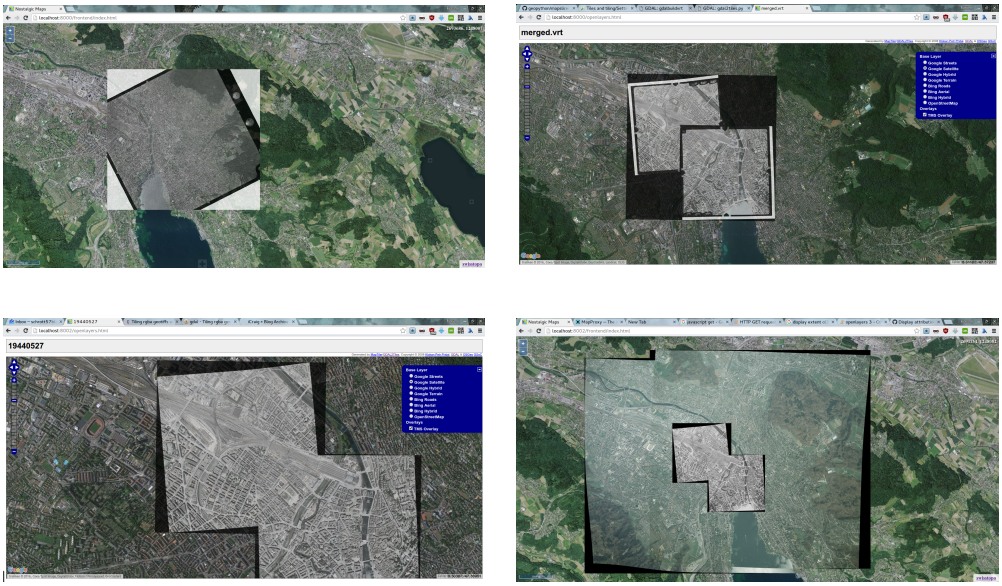


Figure 3.2: Some screenshots of the prototype

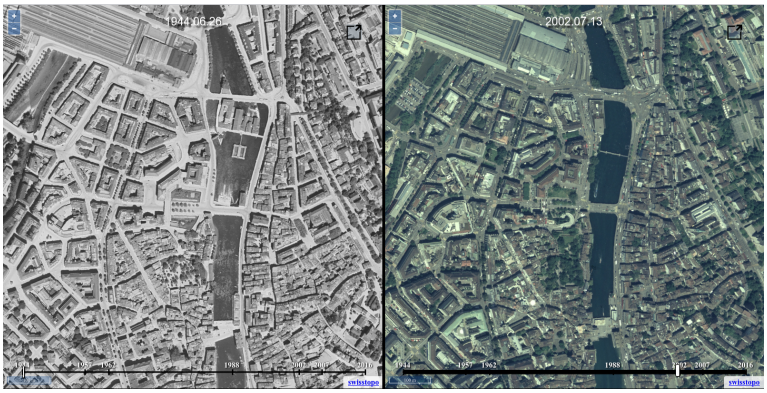


Figure 3.3: The final product

Chapter 4

Results

4.1 Frame Cropping

The presented method for automatically cropping a picture with a black frame works very well with our data. Of all the processed sample data, I could not spot any instance where the cropping would have failed. There might be some examples where it failed, but to have any real statistics we would have to look at each picture manually, which is not practical to do. So, instead of statistics, here are some pictures of rectangles that the algorithm detected:

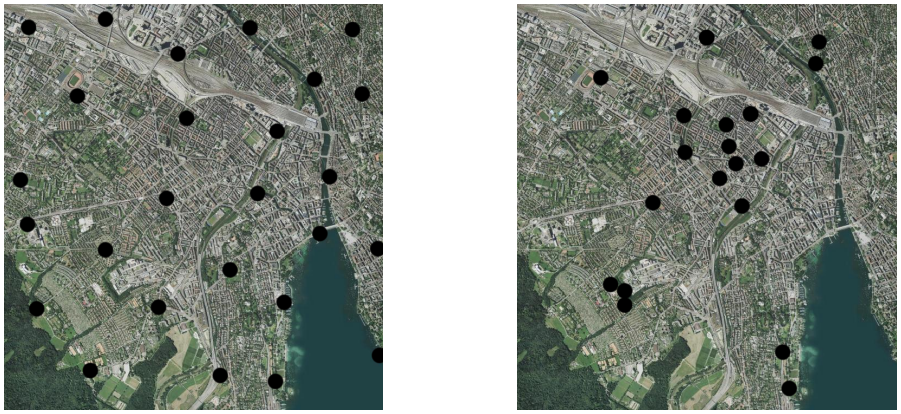
Figure 4.1: The best rectangle to crop away the frame, as found by the algorithm. (Images from [1])

4.2 Feature Based Georeferencing

4.2.1 Manual vs. Automatic Georeferencing

The images generated by the automatic georeferencing process are usually less accurate than the manually georeferenced one. However, most of them are good

enough for our visual purpose. The main difference to the manually georeferenced images is the distribution of the ground control points. These points should ideally be distributed roughly uniformly across the image, with a higher density in areas with large terrain height differences. The points generated by the algorithm are somewhat randomly distributed, with a higher density in the middle of the picture. I suspect that this is because the middle of the picture is least deformed by the perspective camera, making it easier to find matches in the centre. This distribution is far from ideal and leads to pretty bad distortion towards the edge of the image, but it was good enough for our purposes.



(a) Manual (b) Our algorithm

Figure 4.2: A sample of the distribution of matches. (Images from [1])

4.2.2 Did it Work?

Unfortunately, we cannot algorithmically determine whether our results are correct or not. For this, human interaction is still required. Fortunately, we can take a pretty good guess whether it succeeded or not. This comes from the observation that if we do get a correct matching, we usually get at least 15 matches. On the other hand, if our program got it wrong, we would typically only get less than 9 matches. Based on that, a minimum threshold for the number of matches was determined under which we reject the computed solution. Unfortunately, there are still some false positives that are wrongly accepted as a good solution. This can happen if we do get a lot of matches, but only in the very centre of the image. Then, even though our matches are correct, we get a vary badly distorted warped image.

4.2.3 Effects of Flying Height and Time Taken on Success Rate

The algorithm was most successful with recent pictures. (See Figure 4.3). Only a few pictures older than 1950 could be georeferenced using our algorithm. This is caused by various reasons. First, the older images do often show completely changed regions, where a matching is impossible. Even when a region has not changed significantly, the overall picture still looks quite different when we look closely. New trees have grown, others were cut down and river banks have changed.

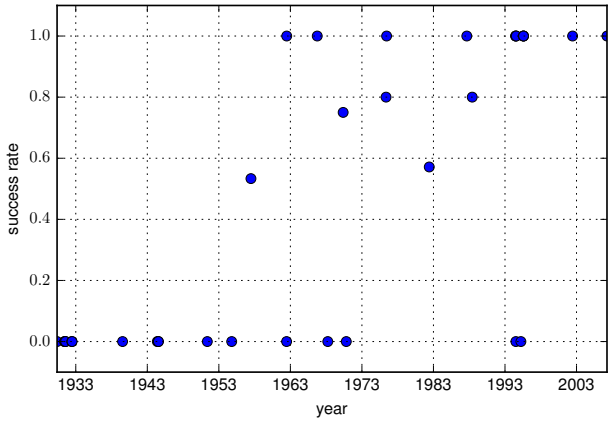


Figure 4.3: Success rate per year

Our algorithm also performs worse if the pictures were taken from close to the ground. (See Figure 4.4). This is because from far above the perspective distortion becomes negligible, while from close-up, it affects the feature descriptors significantly. This makes georeferencing for older images even harder, since many of those were not taken from high up.

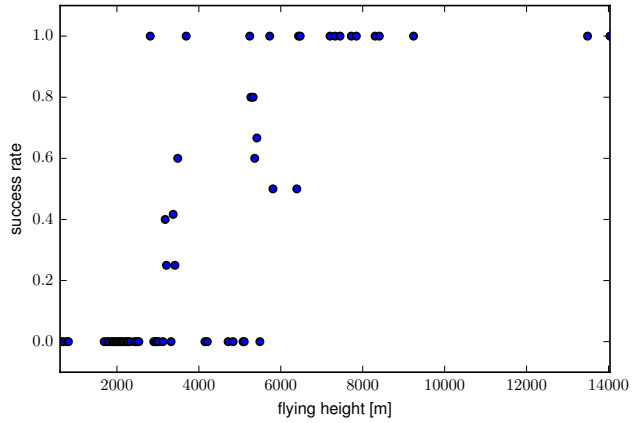


Figure 4.4: Success rate per flying height

There is a correlation between time taken and flying height, as seen in Figure 4.5, so it is not clear whether the time the picture was taken or the flying height is more important to the success of our method. To answer that question with statistical significance, we would need a larger sample.

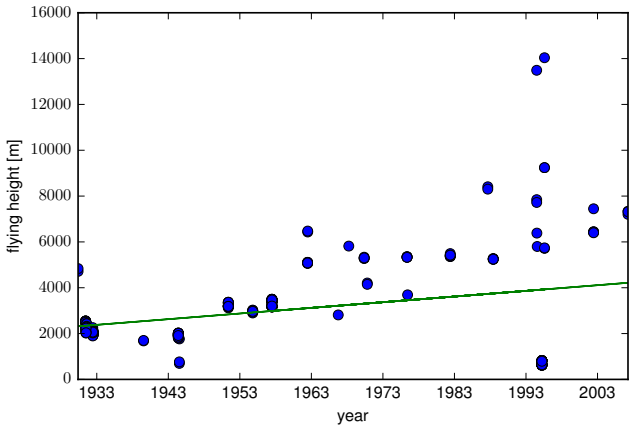


Figure 4.5: Flying height per year

Chapter 5

Conclusion and Outlook

This project started with the goal of creating an interactive map of Switzerland of the past. Accomplishing this goal involved various disciplines. From GIS to computer vision and even frontend web development. We accomplished most of our initial goals and laid down the fundamentals to expand our interactive map even further. We created a script that can perform all necessary steps automatically, from downloading to cropping to georeferencing and finally including the newly produced images in the data used by the frontend such that it can be displayed.

5.1 Outlook

The work that has been done in this project is far from complete. We are still interested in having a system that would allow to grow to the point where the whole earth would be covered in an interactive map of the past. In this chapter, we look at ideas that could help expand the map further and might make the automatic georeferencing process more accurate.

5.1.1 Similarities on the Same Layer

One addition to the algorithm that we used could be to not only try to find matches between the current picture and the one we are trying to reference, but also find similarities on the same layer and on others closer in time. This way, we would have a network of points connected through time. If we now reference one of those pictures, all the others can be updated to get better accuracy. Such an interconnected system would need some way to mark how sure a certain matching is, such that errors would not propagate too far.

5.1.2 Lines and Intersections

A key observation when doing the georeferencing process manually, is that the most stable features over time tend to be roads. An algorithm that would focus on identifying roads and intersections might perform much better than one that tries to find matches between edge features. Such an algorithm might even take the newest vector data of road maps into account. Something similar has already been tried, for instance in papers by Li[32], [33], Zhongliang [55] or Cléri [7].

5.1.3 Layer Merging Using Voronoi Diagram

The way we merged our layers in this project was basically arbitrary. It is entirely possible that a badly matched image will be placed on top of a better matched one, partially hiding the better matched image. To improve this, we could take into account the observation that our algorithm performs better in the centre of the images. The closer we get to the centre, the more matches we get. Therefore, the interpolation in between those matches is more accurate than towards the edges.
So, instead of merging the images by stacking them in an arbitrary order on top of each other, we could merge them in a way that would minimize the average distance to the closest centre of a picture. To accomplish this, we could construct a Voronoi diagram with the centre coordinates of our images. Then, we could cut out our images along the resulting lines of the Voronoi diagram, as illustrated below.

5.1.4 Layer Merging Using Image Stitching

To better merge the images from the same date, we could use image stitching techniques, similar to those used in merging panorama images. This way, we can get seamlessly merged images. For this, the efforts from last year's thesis by Müller [38] could be combined with our approach.

5.1.5 Orthorectification Using Digital Elevation Model

Even if we seamlessly stitch our images together, we would not get as good looking results as the current orthophotos provided by Swisstopo. What is still missing is known as orthorectification. This final step would correct out the perspective distortion caused by the camera. To do this, we would need the digital terrain model of Switzerland. Then, we can project our image onto that 3D model and render a perfectly orthogonal image. For this, we need to know the intrinsic and extrinsic parameters of the camera, such as focal length, position, rotation and zoom among others. These parameters can be approximated using

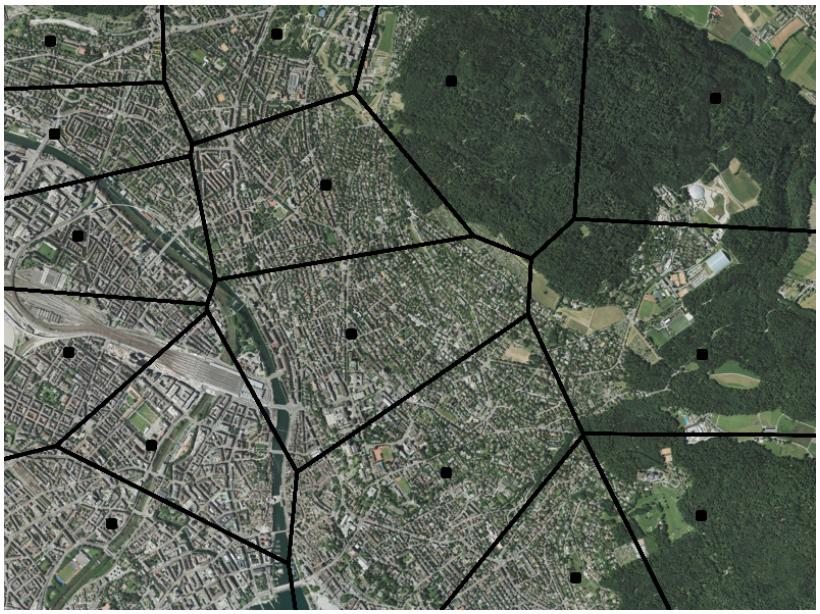


Figure 5.1: A Voronoi diagram constructed from the centre points of our pictures. (Background image from [1])

our mapping from pixel coordinates to geocoordinates. In fact, these parameters are already produced by the employed RANSAC algorithm, but for this project we did not make use of them.

5.1.6 Crowd Sourcing

Even if the automatic process could be improved significantly, there will remain places where human interaction is necessary. This could be to correct mistakes that the system made, add more points to pictures that have a bad distribution of points or even completely georeference a picture that the system could not do. Crowd sourcing would make that manual process practical. We could add various controls to our website such as voting on georeferenced pictures, adding new ground control points and so on. This way, we could create a platform that would grow larger with its users.

References

[3] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. "Surf: Speeded up robust features". In: *Computer vision–ECCV 2006*. Springer, 2006, pp. 404–417.

[4] F Bookstein. "Thin-Plate Splines and the decomposition of deformation". In: *IEEE Trans. Patt. Anal. Mach. Intell* 10 (1988).

[5] Matthew Brown and David G Lowe. "Automatic panoramic image stitching using invariant features". In: *International journal of computer vision* 74.1 (2007).

[6] Peter Bunting, Frédéric Labrosse, and Richard Lucas. "A multi-resolution area-based technique for automatic multi-modal image registration". In: *Image and Vision Computing* 28.8 (2010), pp. 1203–1219.

[7] I Cléri, M Pierrot-Deseilligny, and B Vallet. "Automatic Georeferencing of a Heritage of old analog aerial Photographs". In: *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 2.3 (2014), p. 33.

[10] Martin A Fischler and Robert C Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography". In: *Communications of the ACM* 24.6 (1981), pp. 381–395.

[16] Amanda Geniviva, Jason Faulring, and Carl Salvaggio. "Automatic georeferencing of imagery from high-resolution, low-altitude, low-cost aerial platforms". In: *SPIE Defense+ Security*. International Society for Optics and Photonics. 2014, pp. 90890D–90890D.

[29] Jae Sung Kim, Christopher C Miller, and James Bethel. "Automated Georeferencing of Historic Aerial Photography". In: *Journal of Terrestrial Observation* 2.1 (2010), p. 6.

[31] Stefan Leutenegger, Margarita Chli, and Roland Y Siegwart. "BRISK: Binary robust invariant scalable keypoints". In: *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE. 2011, pp. 2548–2555.

[32] Yan Li and Ronald Briggs. "Automated georeferencing based on topological point pattern matching". In: *The International Symposium on Automated Cartography (AutoCarto), Vancouver, WA*. 2006.

[33] Yan Li and Ronald Briggs. "Automatic extraction of roads from high resolution aerial and satellite images with heavy noise". In: *World Academy of Science, Engineering and Technology* 54 (2009), pp. 416–422.

[34] David G Lowe. "Object recognition from local scale-invariant features". In: *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*. Vol. 2. Ieee. 1999, pp. 1150–1157.

[36] Joan Maso, Keith Pomakis, and Nuria Julia. "OpenGIS web map tile service implementation standard". In: *Open Geospatial Consortium Inc* (2010).

[37] Joan Maso, Keith Pomakis, and Nuria Julia. "OpenGIS web map tile service implementation standard". In: *Open Geospatial Consortium Inc* (2010), pp. 04–06.

[38] Marc Müller. "Mapping the Past". In: (2015).

[41] Stephen M Pizer et al. "Adaptive histogram equalization and its variations". In: *Computer vision, graphics, and image processing* 39.3 (1987), pp. 355–368.

[46] Niles Ritter et al. "GeoTIFF format specification GeoTIFF revision 1.0". In: *URL: http://www. remotesensing. org/geotiff/spec/geotiffhome. html* (2000).

[47] Edward Rosten and Tom Drummond. "Fusing points and lines for high performance tracking." In: *IEEE International Conference on Computer Vision*. Vol. 2. 2005, pp. 1508–1511. DOI: 10.1109/ICCV.2005.104. URL: http://www.coxphysics.com/work/rosten_2005_tracking.pdf.

[48] Edward Rosten and Tom Drummond. "Machine learning for high-speed corner detection". In: *European Conference on Computer Vision*. Vol. 1. 2006, pp. 430–443. DOI: 10.1007/11744023_34. URL: http://www.coxphysics.com/work/rosten_2006_machine.pdf.

[50] Ethan Rublee et al. "ORB: an efficient alternative to SIFT or SURF". In: *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE. 2011, pp. 2564–2571.

[52] Sérgio Ricardo da Silva Santos. "Feature Extraction and Matching Methods and Software for UAV Aerial Photogrammetric Imagery". In: (2013).

[55] Fu Zhongliang and Sun Zhiqun. "An algorithm of straight line features matching on aerial imagery". In: *The Int. Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 37 (2008), pp. 97–102.

[56] Barbara Zitova and Jan Flusser. "Image registration methods: a survey". In: *Image and vision computing* 21.11 (2003), pp. 977–1000.

Web Links

[1] *All aerial images were taken from the Swisstopo LUBIS viewer*. https://map.geo.admin.ch/?topic=swisstopo&layers=ch.swisstopo.lubis-luftbilder_schwarzweiss,ch.swisstopo.lubis-luftbilder_farbe&lang=en&bgLayer=ch.swisstopo.pixelkarte-farbe. Accessed: June 13, 2016.

[2] *AngularJS, HTML enhanced for web apps!* https://angularjs.org/. Accessed: June 13, 2016.

[8] *Elm, a type inferred, functional reactive language that compiles to HTML, CSS, and JavaScript*. http://elm-lang.org/. Accessed: June 13, 2016.

[9] *Example metadata*. https://api3.geo.admin.ch/rest/services/swisstopo/MapServer/ch.swisstopo.lubis-luftbilder_schwarzweiss/19320480121010/extendedHtmlPopup?lang=en. Accessed: June 13, 2016.

[11] *GDAL - Geospatial Data Abstraction Library*. http://www.gdal.org/. Accessed: June 13, 2016.

[12] *GDAL Virtual Format*. http://www.gdal.org/gdal_vrttut.html. Accessed: June 13, 2016.

[13] *gdal2tiles.py, generates directory with TMS tiles, KMLs and simple web viewers*. http://www.gdal.org/gdal2tiles.html. Accessed: June 13, 2016.

[14] *gdal_translate, converts raster data between different formats*. http://www.gdal.org/gdal_translate.html. Accessed: June 13, 2016.

[15] *gdalwarp, image reprojection and warping utility*. http://www.gdal.org/gdalwarp.html. Accessed: June 13, 2016.

[17] *Georeferencing a raster automatically*. http://desktop.arcgis.com/en/arcmap/10.3/manage-data/raster-and-images/georeferencing-a-raster-automatically.htm. Accessed: June 13, 2016.

[18] *GIMP, GNU Image Manipulation Program, The Free & Open Source Image Editor*. https://www.gimp.org/. Accessed: June 13, 2016.

[19] *Google Maps JavaScript API*. https://developers.google.com/maps/documentation/javascript/. Accessed: June 13, 2016.

[20] *GRASS GIS, Geographic Resources Analysis Support System*. https://grass.osgeo.org/. Accessed: June 13, 2016.

[21] *Historic Geodata, Key data and features*. http://www.swisstopo.admin.ch/internet/swisstopo/de/home/topics/geodata/historic_geodata/im_coll/key_dat.html. Accessed: June 13, 2016.

[22] *Image from*. http://gis.depaul.edu/shwang/teaching/geog258/lec7_files/image009.jpg. Accessed: June 13, 2016.

[23] *Image from*. http://elte.prompt.hu/sites/default/files/tananyagok/MapGridsAndDatums/images/691cc4f0.jpg. Accessed: June 13, 2016.

[24] *Image from*. http://opencv-python-tutroals.readthedocs.io/en/latest/_images/feature_building.jpg. Accessed: June 13, 2016.

[25] *Image from*. http://www.edwardrosten.com/work/fast.html. Accessed: June 13, 2016.

[26] *Image from*. http://www.ra.ethz.ch/cdstore/www6/technical/paper130/paper130.html. Accessed: June 13, 2016.

[27] *Imagemagick, montage tool*. http://www.imagemagick.org/script/montage.php. Accessed: June 13, 2016.

[28] *Information system for aerial photographs / LUBIS-Viewer*. http://map.lubis.admin.ch/. Accessed: June 13, 2016.

[30] *leaflet, an open-source JavaScript library for mobile-friendly interactive maps*. http://leafletjs.com/. Accessed: June 13, 2016.

[35] *Mapbox, Maps for mobile & map*. https://www.mapbox.com/. Accessed: June 13, 2016.

[39] *OpenCV, Open Source Computer Vision*. http://opencv.org/. Accessed: June 13, 2016.

[40] *OpenLayers 3, A high-performance, feature-packed library for all your mapping needs*. http://openlayers.org/. Accessed: June 13, 2016.

[42] *Professional GIS Software*. http://desktop.arcgis.com/en/. Accessed: June 13, 2016.

[43] *Python, official website*. https://www.python.org/. Accessed: June 13, 2016.

[44] *QGIS, A Free and Open Source Geographic Information System*. http://www.qgis.org. Accessed: June 13, 2016.

[45] *React, a JavaScript library for building user interfaces*. https://facebook.github.io/react/. Accessed: June 13, 2016.

[49] *RSGISLib, The Remote Sensing and GIS Software Library*. http://www.rsgislib.org/. Accessed: June 13, 2016.

[51] *Screenshot from*. https://api3.geo.admin.ch/rest/services/swisstopo/MapServer/ch.swisstopo.lubis-luftbilder_schwarzweiss/19512170080860/extendedHtmlPopup?lang=en. Accessed: June 13, 2016.

[53] *Swisstopos REST API.* https://api3.geo.admin.ch/services/sdiservices.html. Accessed: June 13, 2016.

[54] *uDig, User-friendly Desktop Internet GIS, A GIS Framework for Eclipse.* http://udig.refractions.net/. Accessed: June 13, 2016.

Appendix A

Code Example

Python Code for Frame Cropping

```
def find_frame_rectangle(image, width, height):
    inner_rect = [width/2, height/2, width/2+1, height/2+1]
    outer_rect = [0, 0, width-1, height-1]

    for i in range(maxIt):
        rect = get_middle_rect(inner_rect, outer_rect)
        test_precision = abs(outer_rect - inner_rect)

        if (test_precision < 2).all():
            return rect

        left, right, top, bottom = get_pixels_on_sides(image, rect)
        rectangle = [left, top, right, bottom]
        frame_likeliness = [mean(is_frame_line(x)) for x in rectangle]
        is_frame_rect = frame_likeliness > threshold

        for j, is_frame in enumerate(is_frame_rect):
            if is_frame:
                outer_rect[j] = rect[j]
            else:
                inner_rect[j] = rect[j]

    return rect
```