Automatic map stitching

Description	Identifying, cropping and stitching a map from screenshots
Period	Fall 2014
Languages & Libs	Matlab
Tags	Computer Vision, Rendering

Nowadays there are many HTML5-based map services, but typically they don't offer any export functionality. To create a full view of the desired region, one can either zoom out (and lose map details) or take many screenshots of different locations and manually stitch them together. This project can automatically load all stored screenshots, detect the map, crop relevant regions, determine images relative offsets and generate the high-res output with zero configuration from any map service.

The first step is to determine which region of the image is the map and which parts should be ignored. This is achieved by assuming that the only moving part of the page is the map itself (no scrolling of the page, no ads, ...). Under these assumptions it is trivial to create a mask similar to the one at Figure 1. Relevant rows and columns (and thus the rectangular cropping region) can be determined based on it.

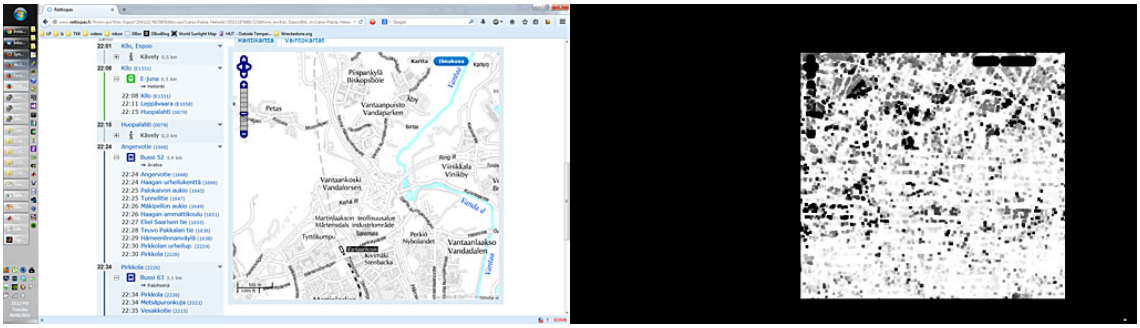

Figure 1: One of the input screenshots on the left and the resulting change-mask on the right. Bright pixels vary between images, and thus can be assumed to be part of the map.

Once map regions have been extracted, their relative positions need to be determined before stitching is possible. This is a special case of a panoramic image generation, which is typically solved by Feature detection and robust RANSAC based matching procedure. However in this map case we only need to estimate the translation in x- and y-directions, and no rotation or scaling should have occurred. We can also assume that the image i should have significant overlap with images $i-1$ and $i+1$, perhaps even with $i-2$ and $i+2$. Fourier transform base Phase correlation is a perfect solution this task, because it isn't computationally too heavy, is very robust, always finds the "best" answer and is very easy to implement. It is based on calculating 2D Fourier transforms G_a and G_b of images a and b and calculating the inverse Fourier transform $F^{-1}((G_a \cdot G_b^*)/|G_a \cdot G_b^*|)$ where element-wise multiplication is used. This should result in a single bright spot, and its mid-point indicates the translation which maximizes the pixel-wise correlation metric.

It is important to apply sufficiently large zero padding when Fourier transforms are being calculated. Otherwise for an image of 1000×1000 resolution the movement of 500 pixels upwards looks similar to 500 pixels downwards, and this ambiguity would need to be resolved via other means.

If image i's location is only calculated relative to $i-1$ and not to $i-2$, a simple chain of position constrains is created which is sufficient to determine each image's location relative to each other. If significant correlation was found also with $i-2$, then an over-constrained linear equation system is constructed, but it has an unique least squares solution.

Once image positions have been determined they need to be composed together. They are processed sequentially one-by-one, and the pixel variation map (see Figure 1) is used to prioritize overlapping pixels. This removes UI elements such as navigation from the final output when possible. The algorithm have been tested with various screenshot sequences, but always using the same configuration and thresholds. Results can be seen in figures 2 to 5, and they all worked out without any problems. The image top shows individual captured maps and the bottom shows the rendered final output (with enhanced image borders for better visualization).

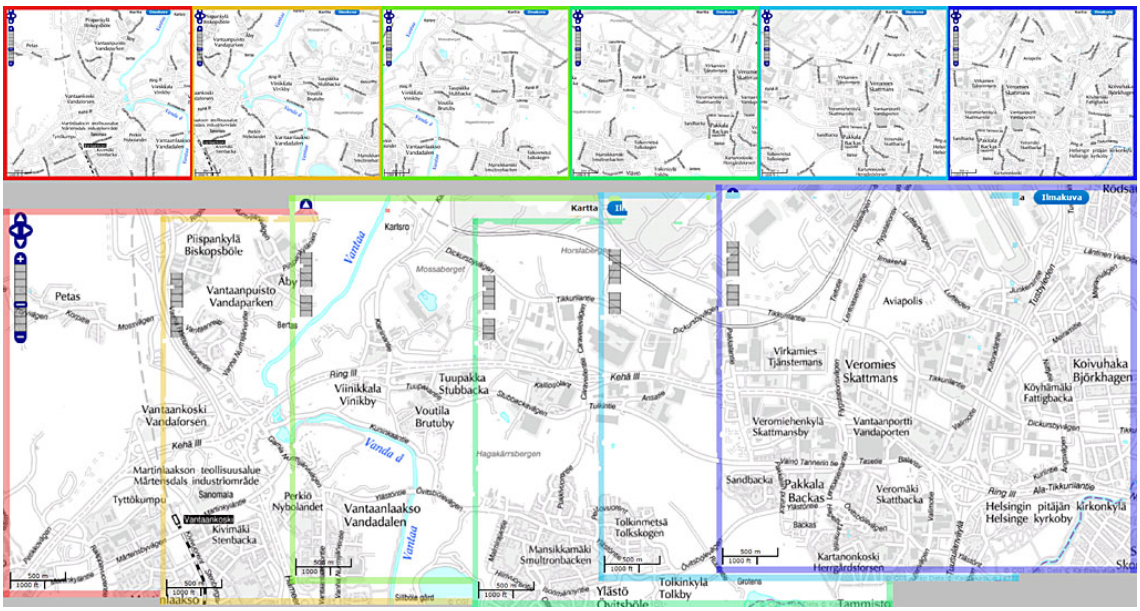

Figure 2: Map parts and the final output from Reittiopas.fi.

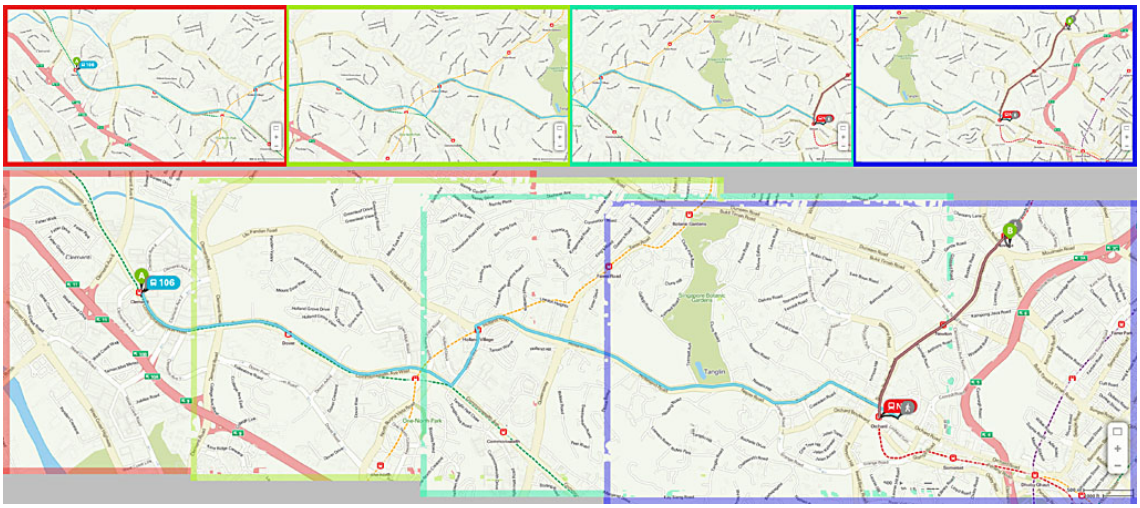

Figure 3: Map parts and the final output from GoThere.sg.

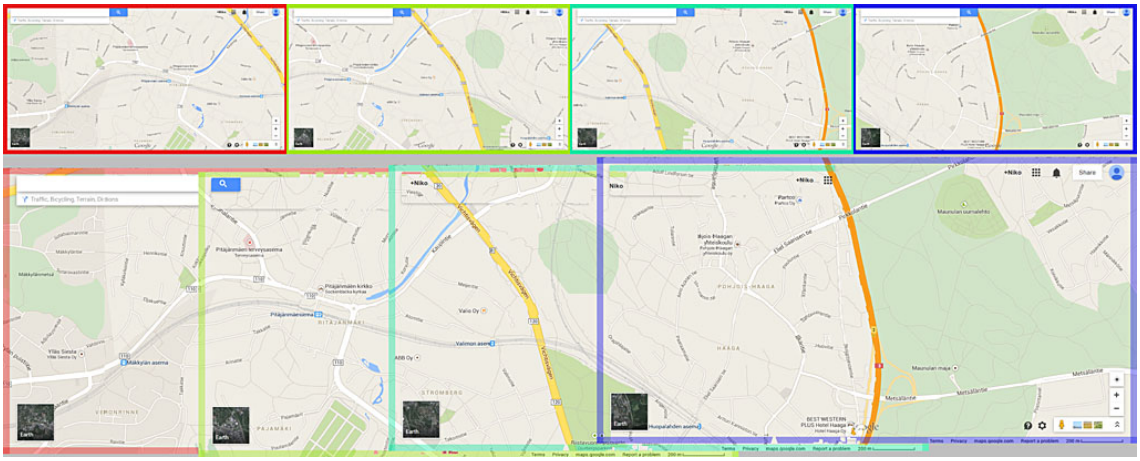

Figure 4: Map parts and the final output from Goole Maps (map view).

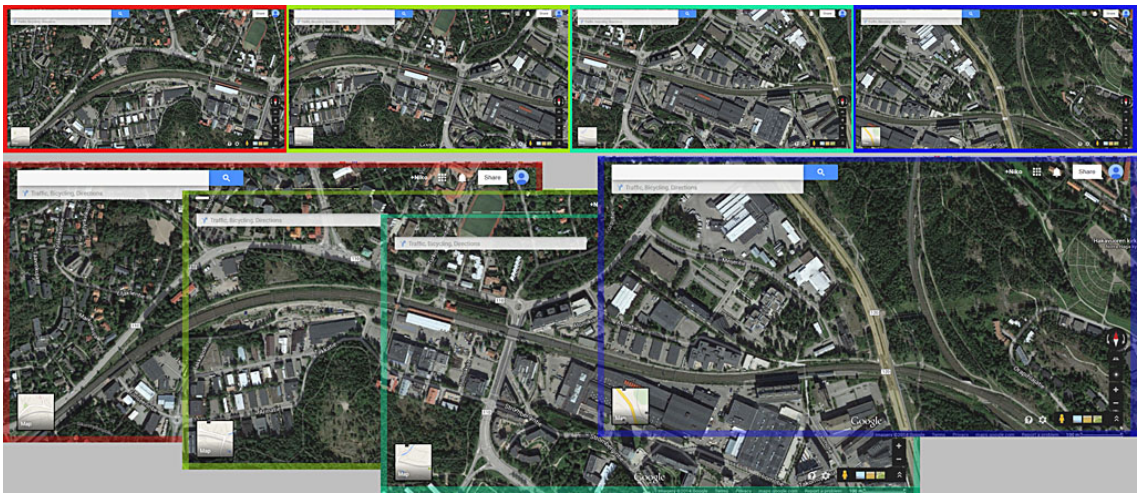

Figure 5: Map parts and the final output from Goole Maps (satellite view).

The only remaining part would be to re-implement all this in C++ (by utilizing FFTW and FastCGI again) and publish it as a web service. Users would either use a browser plugin to capture screenshots or manually save them on the disk and upload from there. This would be a fairly simple system to do and host it on my home server, but it has a few problems. One is the system stability, uptime and monitoring aspect, because if the FastCGI process crashes for any reason then the site would be effectively down.

The other possible problem is the bandwidth and CPU usage. Either almost nobody uses the service which wouldn't be too motivating, or too many people would try to use it simultaneously and it would be too slow or unstable. I might do this one day, but now I'm already having new projects in mind.