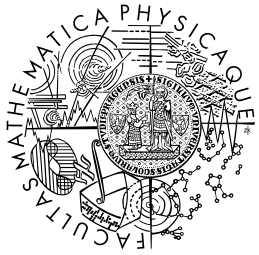

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Peter Šípoš

Application for Automatic Recognition of Textures in Map Data

Department of Software Engineering

Supervisor of the master thesis:	doc. RNDr. Tomáš Skopal, Ph.D.
Study programme:	Informatics
Study branch:	Software Systems

Prague 2018

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague, July 20th 2018 signature of the author

I would like to gratefully thank you for my family who provided support during my study.

Title: Application for Automatic Recognition of Textures in Map Data

Author: Bc. Peter Šípoš

Department: Department of Software Engineering

Supervisor: doc. RNDr. Tomáš Skopal, Ph.D., Department of Software Engineering

Abstract: This work has aimed to implement an easy-to-use application which can be used to navigate through aerial imagery, assign sections of this image for different classes. Based on these category assignments the application can autonomously assign categories to so-far unknown fields, hence it helps the user in further classification. The output of the application is an index file, which can serve as underlying dataset for further analysis of a given area from geographic or economic point-of-view. To fulfil this task the program uses standard MPEG-7 descriptors to perform the feature extraction upon which the classification relies.

Keywords: feature extraction aerial imager image retrieval

Contents

1. Introduction

With the proliferation of online map services such as Google Maps or mapy.cz, the browsing of maps became much more easier for the masses. We can easily search for an address on our computer or mobile phone.

This work has been aimed to use such map services to provide an easy way to annotate and classificate these map tiles.

2. Descriptors

Desriptors simplify image processing by capturing some specific property of the input image. When we extract a descriptor, we transform the image from image space to descriptor space, where certain tasks can performed more efficiently.

2.1 Image representation

Computers store discrete values for each pair of horizontal and vertical coordinate, where each value can be a number for grey-scale images or most commonly a vector of three numbers for each chroma channel. The following equations show the image functions for greyscale and colour pictures:

$$Imagegreyscale\,(x, y) = l$$

where l is the luminance level, and

$$Imagecolour\,(x, y) = (a, b, c)$$

where (a, b, c) most commonly can be $(luminance, u, v)$ or $(red, green, blue)$

2.2 Image comparison

When we compare two images, theoretically we can do it by comparing each pixel from one image to each one in the second image. However, this process doesn't work if the images have different dimensions, some transformation changed the location of pixels e.g. mirroring or the pixel values have changed e.g. different brightness, changed white balance.

Descriptors help to overcome these problems by computing a representation, which is not affected by certain changes in the image. We can create a simple descriptor which computes the average luminance in a greyscale image:

$$averageluminance\,(img) = \frac{1}{width \times height} \times \sum_{x=1}^{height} \sum_{y=1}^{width} (I\,(x, y))$$

The abocve descriptor can be called invariant to mirroring, since the average will not change regardless in which order we walk through the pixels.

2.3 Descriptors in the Application

Currently there are four descriptors in the application, which can be used to classificate each subtile:

1. Mean Luminance - averages the brightness component of images' pixels
2. Mean Hue - averages the Hue component of images' pixels
3. Homogenous Texture - calculates a 62 component MPEG7 Homogenous Texture descriptor

4. Scalable Colour Descriptor - calculates a 32 component MPEG7 Scalable Colour Descriptor

In the next chapter we describe some metrics which can be used to compare these descriptors.

3. Metrics

We have the representation of our images in a restricted domain space and they are representated by a vector of number. Metrics can tell us how close these data are to each other by calculating a number for each pair of vectors.

3.1 Manhattan distance - L_1-distance

This metric borrowed its name from perpendicular streets of Manhattan where we can't go along diagonal shortcuts [1]. This metric calculates how close are two points if we were only allowed to move between them up-and-down and left-and-right and it can be calculated with the following function:

$$L_1(x, y) = |x_1 - y_1| + |x_2 - y_2| + \ldots + |x_n - y_n| \tag{3.1}$$

.

Naturally we can write it down in a more compact form:

$$L_1(x, y) = \sum_{i=0}^{n} |x_i - y_i| \tag{3.2}$$

.

3.2 Euclidean distance - $L_2 - distance$

The shortest path is the straight line - says the proverb and in this case might refer to Euclidean distance. If we connect two points with a line then the length of the section between the points is the value of the Euclidean distance.

$$L_2(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \ldots + (x_n - y_n)^2} \tag{3.3}$$

3.3 k-Means Clustering

The algorithm for a given number k and metric similarility groups the input points into k-groups. Each such group has a center point and by summarizing the distance of points within each group we can calculate the compactness of each group.

3.4 Comparison of metric similarilities

In this experiment we compare the metric similarilities by how compact are the groups created with k-Means clustering and the given similarility.

[1] Although, technically not all streets are perfectly perpendicular e.g. the famous Times Square is more like a bow-tie shape due to the Broadway

4. User Guide

The main application is called APR.Main, which can be used to annotate unknown tiles and shows the result of tile classification when we have enough know data.

Figure 4.1 shows the main window after startup. At this point it can't classificate any tiles, first, we need to setup a distance function by clicking on Settings button. When finished by clicking on the Update settings button we set it as the current metric Figure 4.2.

By clicking on Auto classification button, we can don't need to click on each tile by one Figure 4.3 .

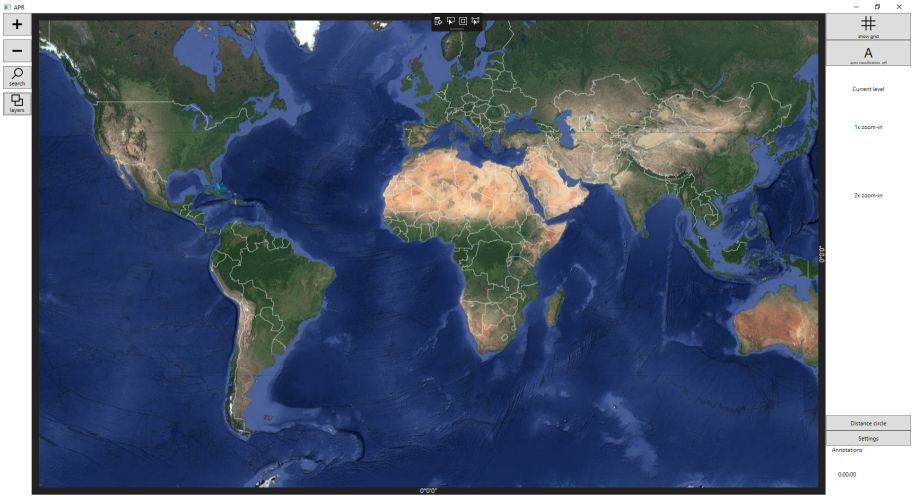

Figure 4.1: The APR's main window after startup

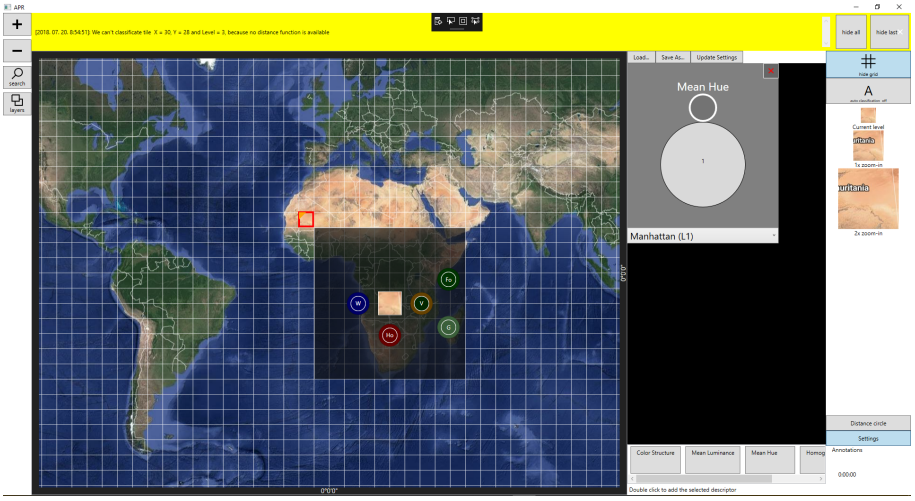

Figure 4.2: Setting the current metric for classification

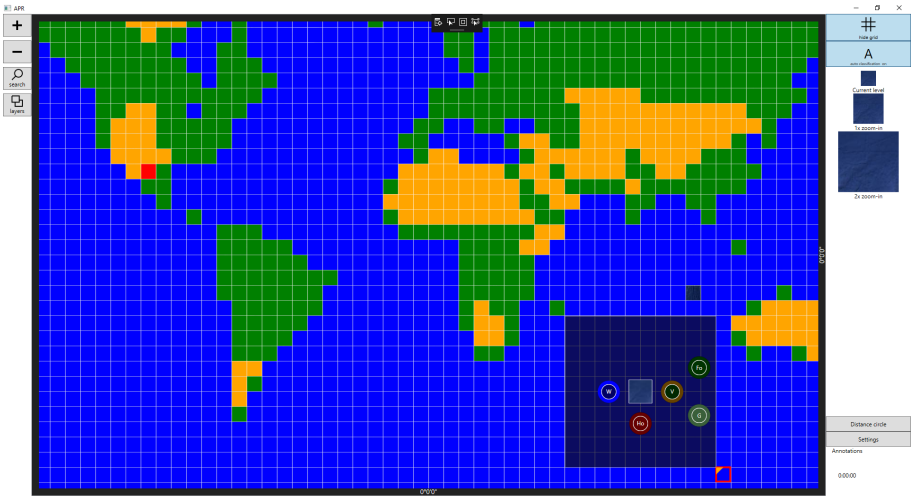

Figure 4.3: Result of the run of automatic classification

5. Programming Guide

In this chapter I would like to describe how each part of the application contributes to the main functionality. The primary goal during implementation was to improve the usability of a classic desktop application on touch enabled Windows-based tablets. Although it was attempted to implement a secondary application for UWP.

The source code for the application is attached in Attachement A.1 .

5.1 Main Application Window

Each section of the main application windows is backed with its own view model. These particular view models help to separate the logic into multiple classes even though there is only one window.

5.2 Tile Loading

There exist components which can show map from some specific vendor. However, it was decided to implement a custom map control, so it is possible to freely access downloaded image data and save to speed up navigation and reuse it for classification.

There are two main components for tile loading: the tile services and the map control with tile layers. Tile services are tailored for different web-based map providers such as Google Maps, Bing Maps or mapping services of mapy.cz. The main assumption for these services is that for each triplet of level, horizontal coordinate and vertical coordinate they return an image representing that particular area. Sometimes these services change without any notice and that is why generation of own image tiles turned out to be helpful.

The map control with tile layers transforms the user interaction on screen to map movements and based on these movement it orchestrates tile loading and can trigger automatic classification of newly added tiles. Furthermore, we have other objects on the map - annotation circle, pins, coordinate status, which need to follow the map movements too.

When working with tiles we need a way to uniquely identify each of these tiles. We can distinguish three types of coordinates when positioning tiles in the horizontal plane:

1. mozaic coordinate - each tile is one step in this coordinate system - starting from zero

2. pixel coordinate - each pixel of the tiles is one step in this coordinate system - we can convert these two as follows :

$$pixelcoor = mozaiccoor \times tilewidth$$

   , where the tile width is usually 256 pixels in both vertical and horizontal directions.

3. geographical coordinate - the classical latitude, longitude pair to pinpoint a place on Earth's surface out of many coordinate systems. OpenStreetMaps's Nominatim address search system - similarly to most web services - uses WGS84 coordinate system[OSM18]. We can refer to map's size in pixels on one particular level as virtual width or height and the conversion from WGS84 to pixel coordinates is as follows:

$$x = 0.5 \times \lambda/\pi \times virtualwidth \tag{5.1}$$

$$y = 0.5 \times \log\left(\tan\left(\frac{\pi}{4} + \frac{\sigma}{2}\right)\right) \times virtualheight \tag{5.2}$$

. The above transformation is called Mercator-projection in honour of its inventor Gerardus Mercator[Wik18]

The main problem with map tiles is that there is quite many of them. On starting level we have one tile which is 256x256 pixel large and both dimensions grows exponentially as we zoom in following $Side(level) = 2^{level} \times 256; level = 0, 1, 2, \ldots, 20$.

To avoid performance problems only those tiles are loaded which are in view and getting disposed when they are moved out of view. Each tile is given a key similar to Z-index and when the map is moved two lists of these indexes are created for tiles to be added or deleted.

Code for this functionality can be seen in Code 5.1, where the leftkey, rightkey, topkey and bottomkey variables are the visibility borders in mozaic coordinates and ordDisplayed, ordVisible are the are lists of key, tile pairs which are displayed and should be displayed ordered by the keys.

```
int visIndex = 0;
int disIndex = 0;

while ((disIndex < ordDisplayed.Count)
        && (visIndex < ordVisible.Count))
{
    ZKey visibleKey = ordVisible[visIndex];
    ZKey displayedKey = ordDisplayed[disIndex].Item1;
    UIElement tile = ordDisplayed[disIndex].Item2;

    // Should be visible, but it's not displayed, yet
    if (visibleKey < displayedKey)
    {
        request.Add(visibleKey);
        visIndex++;
    }
    // Precedes the current visible tile, but
                // we haven't found it in the visible list
    // We should remove it
    else if (visibleKey > displayedKey)
    {
        removal.Add(tile);
```

```
            disIndex++;
        }
        // Found a displayed, which should be visible
                    // All good, move along
        else
        {
            visIndex++;
            disIndex++;
        }

}

// Add which should be visible
while (visIndex < ordVisible.Count)
{
     ZKey visibleKey=ordVisible[visIndex];
     request.Add(visibleKey);
     visIndex++;
}

// Remove that shouldn't be visible
while (disIndex < ordDisplayed.Count)
{
     UIElement tile=ordDisplayed[disIndex++].Item2;
     removal.Add(tile);
     disIndex++;
}
```

Code 5.1: Calculating which tile is within the map's child elements

As you can see we are checking the indices of the already existing tiles against the precalculated "'imaginary"' tiles list. The ZKey serves as a simple representation of the tile location calculated as $key = ((ux << 32)|uy)$ where ux and uy are the horizontal and vertical mozaic coordinates.

5.3 Description Creation

Descriptors are implemented in Descriptors.CLR project which depends on the API provided by the Measures.PCL. Every descriptor is represented by an array of floating point numbers which is calculated from the image data passed in the Extract function. We are using data in the extract function so we can pass the needed information for every possible descriptor, while if we used e.g. pixel data that possibly would be incompatible with some descriptor.

To extend the existing list of descriptors the following changes should be done:

1. Descriptors.CLR/NewDescriptor - add the implementation here, preferably to the APR.Measures.Descriptors namespace

2. ExtractionInterop.CLR/ExtractionInterop (optional) - if the new descriptor is a native one, add the code for its invoking here

3. DescriptorViewModel.CLR/DescriptorStringModel - add viewmodel for it which will show the options for the given descriptor

4. DescriptorViewModel.CLR/ConfigurationConverter - add another if-else branch for the newly created descriptor object

5. DescriptorViewModel.CLR/ConfigurationGraph - in ConfigurationGraph also instantiate the DescriptorSelector and add it to the available descriptors' list

5.4 Classification

The map tiles' dimensions are 256x256 and are retrieved as a single file. While this size provides a good compromise between number of files in the viewport and size of one individual image, from classification point of view they still can contain multiple objects. Hence, each image tile is divided by an 8x8 grid to 64 sub images which are individually processed.

The classification algorithms are implemented in the APR.Main project, in the APR.Main.Classification namespace. Currently two classificators are available:

1. Classificator - the category of the new comer is determined by the tag of the closest neighbour

2. KNearestClassificator - a given number of kclosest neighbours vote

While the application is running it maintains a cache of classifications, which is retained in memory until the application exists. This should be replaced with a better logic for invalidating the classes in the cache.

Conclusion

While this application does not serve its main purpose - serving as a decent Master Thesis Project - at least it is usable as a map browser. It is hard to draw conclusion from such a short work. This will teach me hard lesson how important is communication, time-managing and probably the most important one over-estimating one's abilities is a big mistake.

Cannot believe what I was thinking when continued postponing the meetings with my supervisor. When I could not make a decision hoped that the problem can solved for next week, but as expected I digged deeper in my misery and later I was too ashamed to show up and as expected could not fix this mess all by alone. Also this inability of decision making made start developing three version of the same application, but all of them incomplete.

Please consider this work as a way of submitting the application. Sadly, I cannot state it is a Master Thesis, since it is not.

6. Appendix A

6.1 Custom Map Tile Generation

The main disadvantage of using web based tile services like Google Maps is that there are prone to change or they doesn't reflect the area under our expected conditions.

6.2 Converting

We can download some custom aerial pictures from Prague OpenData[1]. After that it can be converted to map tiles by gdal2tiles, which can be seen in action on Figure 6.1

6.3 Application with Custom Aerial Imagery

Editing Config/custom_layers.txt file, we can point the APR to use these custom tiles. One view for such a usecase is visible on Figure 6.2 for dataset which is included as Attachement A.2

[1] Prague OpenData[Pra18]

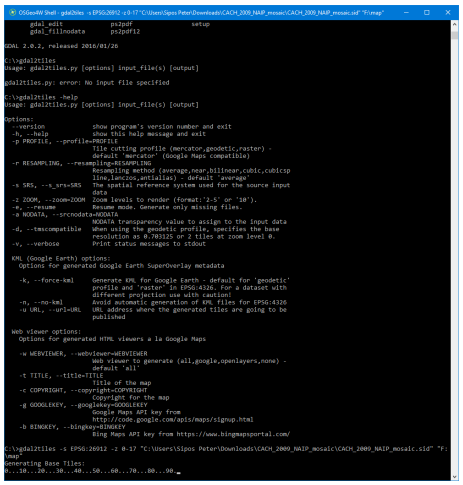

Figure 6.1: Running Gdal2Tiles command to generate the slippy tiles

Figure 6.2: View on APR which uses a custom tile set

7. Appendix B

7.1 Compiling OpenCV libraries

Binaries for OpenCV can be downloaded from the official https://opencv.org/ [1] website, but in some cases the pre-built files might not suit our needs. In this particular case I'd like to show the compilation process of the OpenCV World package for 32-bit architecture for UWP.

During the following steps we create a Visual Studio Solution from source files which can be used to build the desired libraries. This example was performed using Visual Studio 2015 Update 3, but it should be possible to achieve the same results with any recent version.

1. Download and install CMake (version 3.8.2 in my case) 64-bit Windows Installer from "https://cmake.org/files/v3.8/cmake-3.8.2-win64-x64.msi"[Pro18a]

2. Download OpenCV sources as ZIP files from the following sources: OpenCV[2] and OpenCV Contrib [3]. Alternatively, it can be downloaded directly from OpenCV under Releases: OpenCV Releases] [4]

3. Extract files from both files to a common directory. While extracting skip colliding files - of which aren't many, just the common files for Github repositiories.

4. Add source and build folder to CMake as seen on Figure 7.1

5. Click Configure and select Visual Studio 2015 (Figure 7.2)

6. Selecting the needed components and by finishing the wizard, CMake creates a Visual Studio solution for compilation.

[1] [Pro18b]
[2] https://github.com/opencv/opencv
[3] https://github.com/opencv/opencv_contrib
[4] http://opencv.org/releases.html

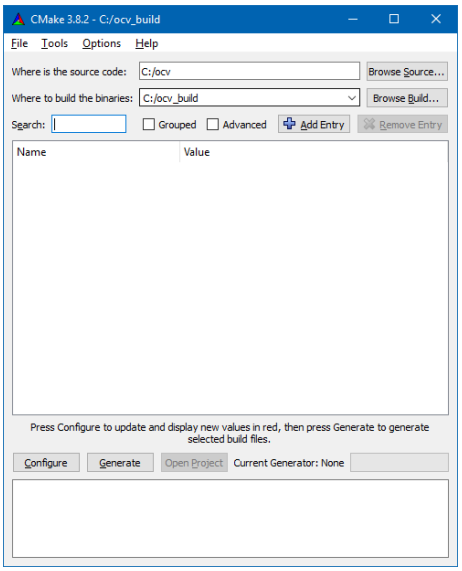

Figure 7.1: OpenCV source and build location

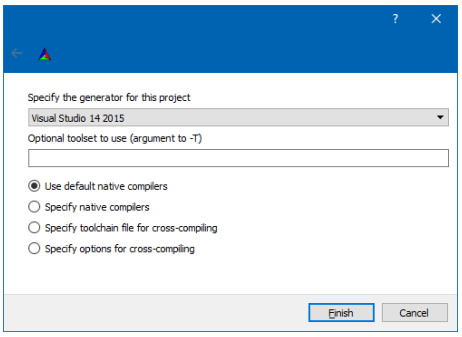

Figure 7.2: CMake configuration dialog

Bibliography

[OSM18] Open Street Map OSM. Open street map wiki - wgs 84. https://wiki.openstreetmap.org/wiki/Converting_to_WGS84/, 2018. [Online; accessed July 20th, 2018].

[Pra18] Prague. Opendata website. http://opendata.praha.eu/dataset/, 2018. [Online; accessed July 20th, 2018].

[Pro18a] CMake Project. Cmake project website. https://cmake.org/, 2000 - 2018. [Online; accessed July 18th 2018].

[Pro18b] OpenCV Project. Opencv project website. https://opencv.org/, 2000 - 2018. [Online; accessed July 18th 2018].

[Wik18] Wikipedia. Mercator projection. https://en.wikipedia.org/wiki/Mercator_projection/, 2018. [Online; accessed July 20th, 2018].

List of Figures

List of Tables

List of Abbreviations

UWP Universal Windows Platform. 6, 11

A. Attachments

A.1 Source code for the application

A.2 Example dataset