Map of Denmark

First-Year Project, Bachelor in Software Development,
IT Univ. of Copenhagen

Group 12
Jakob Melnyk jmel@itu.dk
Niklas Hansen nikl@itu.dk
Emil Juul Jacobsen ejuu@itu.dk
Jens Dahl Møllerhøj jdmo@itu.dk

Supervisors: Lars Birkedal
Advisors: Jonas Brabrand Jensen and Filip Sieczkowski

May 25th, 2011

Contents

Chapter 1

Preface

Chapter 2

Background

2.1 Problem area

Over the last decade people have switched from traditional roadmaps to using the web-maps. This is a change without any negative side-effects. The online services remove all the problems with determining the quickest route between two points and you spend no time browsing the pages of the map to find what you need. With the popular smartphones the online map is even more useful, since you no longer need to prepare your trip before you leave.

The online maps have now been used for many years and haven‘t been slow at adopting new features to improve their usability. They have both implemented satellite-maps that allow us to browse the entire planet from above, and lately the feature called Google Street View has upped the stakes when allowing us to look at any direction from a given point of a road. The two maps that we use the most are Google Maps and the Danish map called Krak. These maps both have the mentioned features but slight differences in the way the user navigates and searches for routes.

Because of the widespread knowledge of the online maps, the users have been accustomed to certain features and ways of using the map. It is very important that we, with a new map program, use this knowledge to our advantage and don‘t try to reinvent the wheel. By using some of the commonly used controls in our map, a user will be able to quickly adapt to our program and use it efficiently.

2.2 Requirements for the map

2.3 Our requirements

2.3.1 Project requirements

2.3.2 Our own requirements

2.4 Data set

We have been provided with a dataset of roads and intersections in Denmark from Krak. Additionally we got some code for loading the data in from the text files. We have only made minor changes to the code for loading the data.

2.4.1 UTM-coordinates

It is important to note that the KrakNodes are in UTM-32 coordinates. When using the UTM standard the origo is placed at the south-west corner. These coordinates need some conversion when using in Java since the origo is placed differently.

2.4.2 Graph

When the data has been loaded it is stored as a Graph containing KrakNodes and KrakEdges. The KrakEdges are the road segments and contains the name of the road, an estimated drive time, a direction of traffic and references to the two KrakNodes that are at either end of the road. The KrakNode itself contains only the coordinates for the point. The Graph itself contains a number of useful methods for searching the data like getting all edges that is connected to a KrakNode. We will be using these methods extensively throughout the project both for drawing the map and for finding the route between two points.

2.5 MVC structure

Chapter 3

User Interface analysis

In this chapter we describe our decisions and present our analysis and arguments regarding some of the features that we find interesting.

3.1 User interface as a whole

When we designed the first version of the graphical user interface in the first part of the project, we decided to make a window inside of the graphical user interface where the actual map should be displayed. We chose to have have this window placed on the right side of our graphical user interface and interaction with the user mainly placed on the left.

We believe that this is a simple way of representing a user interface for a map. A lot of software use a menu bar with dropdown menus for selecting different functions. When we designed our outline for the graphical user interface, we did not design it with a huge amount of functions in mind.

The features that we have implemented in this version can easily fit in our simple user interface, but if features like searching for roads, route planning, etc. are included, then space and overview may become an issue on the left side.

Depending on the feature, we feel it would be beneficial to let the main window change when different feature types are selected.

Below is a screenshot of our standard user interface. How to use it will be explained in the Manual on page 17.

Figure 3.1: Screenshot of GUI

3.2 Interesting features

This section presents some of the interesting features we have implemented.

3.2.1 Zoom

We have a few options for zooming in and out on the map. As described in section 2.3.1 Project requirements on page 6, it was required that we made it possible to zoom by dragging a box around the part of the map the user wants to view.

In addition to the option of using the mouse to zoom, we have implemented a zoom-in and out function on the GUI and a hotkey for zooming out to the original view. We made the original view function a hotkey only because we did not want to have too many buttons on the left side. We considered making it a menu bar function, but we did not manage to get it into this version.

We felt we really needed a zoom out function, so users do not need to close the program and start it again, when the user wants to view the map further zoomed out. A combination of the zoom in and out functions helps the user a lot when navigating the map.

We have limited how far a user can zoom in and out. If the user tries to zoom out further than the original zoom level, the view will default to the original zoom level. If the user attempts to zoom in further than a width or height of 200 in UTM32-coordinates, the zoom function will do nothing. This limitation could be improved by zooming in on the smallest possible zoom at the postion the user selected, instead of doing nothing.

3.2.2 Navigation

We have made it possible for the user to navigate the map by using the arrow buttons on the graphical user interface. When one of the buttons are pressed,

the “view” will move in the direction specified by the button. While it was not specified as a requirement for the project, we felt it was a necessity to implement at least basic navigation functionality.

Like we did with the two zoom functionalities, we have limited how far a user can move around the map. The user is free to move around the map, but if user moves outside the bounds of the map in a way where the view would show an image that is not part of the map, the move function will not do anything.

3.2.3 Hotkeys

We have implemented hotkeys for all the buttons on the graphical user interface plus an additional for zooming back to the original view. When we discussed the benefits of hotkeys, we felt it was important for experienced users of the software should have a less cumbersome time navigating the map.

At first we just had hotkeys for the clearing of markers (mentioned in section 3.2.4) and zooming out to the original view, but we later added the hotkeys for the rest of the functionalities. If more features are added in a future version, it would be important for us that a hotkey were provided, if at all possible.

3.2.4 Route planning and markers

Part of the requirements for the project was to provide the user with a way to get the fastest or shortest route from one point to another. We accomplish this by putting a “marker” at the spot where the user clicks with the mouse. The marker shows which number it is. This will change if a marker is removed. Originally we had “pins” instead of markers, but we changed it, as we felt the pins we had were a bit large.

We have made it possible to place more than the two markers that the project requirements asks for. If the user places more than two markers, the software will find the shortest route between 1-¿2 and 2-¿3 and so on. This was cheap for us to implement, and we felt it added a good feature to our software.

We have implemented two methods of removing pins from the map. We have assigned a hotkey to the graphical user interface button “Clear Markers”, which clears all the markers on the map. The other way of removing pins is by clicking on them. This functionality is both intuitive and confusing at the same time. It is intuitive to click the marker you have just placed if you want to remove it, but it is not obvious in our interface. We believe that it is enough to have the “clear all markers” functionality for those who do not find it intuitive to click markers to remove them, and for the users that do find it intuitive, we offer them an easy way to undo a missclick.

3.2.5 Bike/car

Another interesting feature in our Map of Denmark project is the option to switch between bike and car routes. The user interface will start with car selected when the program start.

If a route is marked by the user, it will display the length and the estimated travel time on the left part of the user interface for a car. When the bike option is selected, it will recalculate the route for a bike without visiting highways and other roads that bikes cannot drive. If the user switches back to the car mode, it will recalculate the route again, but not visit small paths and other roads where a car is not allowed to drive. The estimated travel time is also recalculated. The user does not need to have planned a route before he/she changes the type of route planning.

We have implemented this to help our software target a wider group of people. The bike/car options were a bit costly to implement, but we categorised it as a very beneficial feature and we did not feel we could leave it out.

3.3 Features not implemented

This section presents some of the features we have chosen not to implement. These features are not in the final program, because we did not feel there were compelling arguments for implementing them.

Features that we wanted to implement, but did not make it into the final version will be discussed in chapter Product conclusion on page 18.

3.3.1 Choice of roads to be displayed

3.3.2 Smooth scrolling

3.3.3 Dynamic route finding

Chapter 4

Implementation

This chapter describes how we have implemented some of the more interesting features of the software. We aim to describe it to enough detail that this chapter can serve as a guideline for implementing the functionalities we describe.

4.1 UTM-conversion

When the graphical user interface part of the map tries to communicate with the model through control, some conversions of the different kind of values are necessary. Both when going from coordinates in the java-coordinate system to UTM32-coordinates and back.

We need to convert the values when we want to use the mousezoom and when we want to place the markers for pathfinding. We get an input on the graphical user interface when we mousezoom and this needs to be converted to UTM-coordinates so that we can create the new boundaries of the zoomed rectangle.

When we place markers for pathfinding, we do the same as when we do mouse-zoom, but instead we store the point as UTM32-coordinates and whenever we move the map, we convert it back to pixel-coordinates so that we know where to draw.

The java-coordinates have origo in the top left corner with the y-coordinate increasing the further down the y-axis you go. UTM32-coordinates are a bit different. UTM32 has origo in the bottom left corner and the y-coordinate increasing the further up you go on the y-axis.

We have a utility class with methods for converting the points back and forth. One takes a point from the view, the model and the view itself uses this formula for converting the pixelpoint to the UTM32-point.

a_y = $canvas_{height}$ - a_y

UTM_x = $bounds_x$ + (a_x / $canvas_{width}$) * $bounds_{width}$

UTM_y = $bounds_y$ + (a_y / $canvas_{height}$) * $bounds_{height}$

return point(UTM_x, UTM_y)

Below is an illustration of the conversion from pixel to UTM.

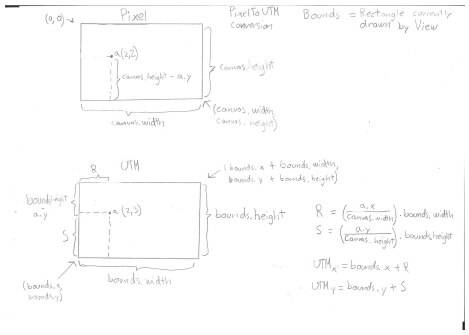

Figure 4.1: Conversion from pixel to UTM

To convert from UTM to pixels we use the same formula but reversed.

$Pixel_x$ = ((a_x - $bounds_x$) / $bounds_{width}$) * $canvas_{width}$

$Pixel_y$ = ((a_y - $bounds_y$) / $bounds_{height}$) * $canvas_{height}$

$Pixel_y$ = $canvas_{height}$ - $Pixel_y$

return point($PIXEL_x$, $PIXEL_y$)

4.2 Mousezoom

4.3 Dijkstra vs A-star

In preparations to implementing the path finding feature to our program, we knew two possible choices. They are named Dijkstra and A* (A-star), and are quite similar but behave a little different.

The Dijkstra algorithm uses a minimum priority queue to find the shortest path from a given node to every other node by looking at the edges connecting nodes. The program will take a node from the priority queue and add all the other nodes that are connected from the current to the priority queue. The priority queue takes a value to the node and this should be the distance to the current node

plus the length of the edge between the two. Since the priority is made to return the node with the smallest value associated with it, the next node in line will always be the one which is closest to the start node. This procedure continues until all nodes have been visited and by logging what edge led to all the nodes it is possible to trace back the route to the start node.

This algorithm is great if need to find the distance from one point to many other points, but can be quite slow since it just searches in all directions without concerns to the direction of the target node.

This is where A* comes in handy. The A* algorithm is a modification to Dijkstra that also looks at the estimated distance from the given node when determining the value for the priority queue. When using the geographical distance as a measure of "best route" the value would be the current distance from the start node plus the direct distance to the target (as if there were a road directly to the target). With this subtle change the algorithm will prioritize nodes that are relatively closer to the target than those that are in the other direction. This makes the algorithm much faster since it will not pay much attention to the roads that are not in the direction of the target. We have decided to use the A* algorithm since we only calculate routes between two distinct nodes and therefore don't need the route from the start node to all others. The time reduction that A* gives is also a definite plus since no user wants to sit and wait too long for the program to find the route.

4.4 Evaluator

In order to make our path finding algorithm flexible enough for different interpretations of the "best route", we have added an entity called Evaluator. This is an object that has the responsibility of evaluating a node relative to the target node. The Evaluator also has the responsibility of calculating the heuristics that the A* algorithm relies on. By using the Evaluator we are able to use the same path finding algorithm for two very different tasks, namely the biking route and the car route. The major difference between these is that the bike uses the distance and the car uses the total drive time. This implementation is also a good example of making our code ready for future features, since if we needed to add other means of transportation or simply variations of the ones we have, we would only need to create new Evaluator objects and not change a single line of code in the A* algorithm.

4.5 Quadtree

4.6 Serialization

4.7 Floats

Chapter 5

UML-diagrams

5.1 MVC

5.2 Simple Diagram

5.3 Control flow

Chapter 6

Tests

6.1 WhiteBox: closestEdge

6.2 JUnit

6.3 System test

Chapter 7

Manual

7.1 Navigation

7.1.1 GUI

7.1.2 Keyboard

7.2 Zoom

7.2.1 GUI

7.2.2 Keyboard

7.3 Route find

7.4 Bike/car

7.5 Resize

7.6 Road display

Chapter 8

Product conclusion

Chapter 9

Group norms

Chapter 10

Diary

3-4

Chapter 11

Worksheets

4-5

Chapter 12

Process description and reflection