Julius-Maximilians-Universität Würzburg
Fakultät für Mathematik und Informatik

Dissertation

Dynamic Label Placement in Practice

Nadine Schwartges

Februar 2015

Betreuer:	Prof. Dr. Alexander Wolff
Gutachter:	PD Dr. Martin Nöllenburg, Karlsruher Institut für Technologie (KIT)

Contents

Summary

This thesis is about labeling *interactive maps* in practice. In computer science, the *labeling problem* corresponds to a packing problem: given a container, pack it as densely as possible with objects from a given set such that each object lies completely within the container and the objects do not overlap each other. Compared to the packing problem, the labeling problem is constrained in that the position of each object, or *label*, is restricted to a certain area within the container. More precisely, the general map-labeling problem is as follows: given a set of geometric objects to be labeled, or *features*, in the plane, and for each feature a set of label positions, or *candidates*, maximize the number of placed labels such that there is at most one label per feature and no two labels overlap. There are three types of features in a map: point, line, and area features. Unfortunately, one cannot expect to find efficient algorithms that solve the labeling problem optimally.

Interactive maps are digital maps that only show a small part of the entire map whereas the user can manipulate the shown part, the *view*, by continuously panning, zooming, rotating, and *tilting* (that is, changing the perspective between a top and a bird view). An example for the application of interactive maps is in navigational devices. Interactive maps are challenging in that the labeling must be updated whenever labels leave the view and, while zooming, the label size must be constant on the screen (which either makes space for further labels or makes labels overlap when zooming in or out, respectively). These updates must be computed in *real time*, that is, the computation must be so fast that the user does not notice that we spend time on the computation. Additionally, labels must not *jump* or *flicker*, that is, labels must not suddenly change their positions or, while zooming out, a vanished label must not appear again.

In this thesis, we present efficient algorithms that dynamically label point and line features in interactive maps. We try to label as many features as possible while we prohibit labels that overlap, jump, and flicker. We have implemented all our approaches and tested them on real-world data. We conclude that our algorithms are indeed real-time capable.

The thesis consists of two parts. The first part deals with the problem of labeling point features. The first solution that we propose is an *offline* solution: our algorithms build a data structure in a preprocessing; at runtime, the labeling is just queried. We focus on the case that the user zooms, that is, when the user changes the *scale* of the shown part of the map. While zooming out, on the screen, points get closer to each other. As we require that labels have constant size on the screen, we must remove some labels in order to avoid overlaps. To ensure that a label does not jump, we center the label at its feature; to ensure that a label does not flicker, to each label, we assign a connected interval of scales in which the label is placed. In order to solve this problem, we present an exact algorithm based on integer linear programming and some heuristics based on greedy strategies. As the exact algorithm has a very high computation time, we only use it in order to verify that our heuristics compute near-optimal solutions.

Next, we compare two *labeling models* for *online* labeling, that is, the labeling is computed on the fly. First, we give an algorithm that makes use of a *slider model*, that is, at runtime, a rectangular label is permitted to move with its lower edge along its corresponding point feature (or *reference point*) in order to make space for other labels. Additionally, we present an algorithm using a *fixed position* model where the reference point lies at the center of the bottom edge of its corresponding label. We conclude that, in the labeling using the slider model compared to the labeling using the fixed-position model, the number of placed labels increases considerably. Further, we apply two methods for detecting overlapping labels: we use a rather naive approach that tests each pair of labels for overlaps and, in order to speed up computations, we use a simple geometric data structure. We found out that the data structure tends to make updates faster, but only slightly.

In the second part of this thesis, we study the problem of labeling streets online. First, we examine the problem of placing labels that follow the curvature of their corresponding streets. Labels should be readable and aesthetically pleasing, for example, labels should have few bends. We present an algorithm that evaluates the aesthetics of each single label position along a street and finally decides for one position.

Labels that follow the curvature of their streets are sometimes hard to read. On the other hand, the legibility of labels along a route that leads the user to a target (as common in navigational devices) is very important. Therefore, we complement the just-mentioned algorithm by an algorithm that places axis-parallel, rectangular labels in an interactive map in a bird view. In this approach, we attach one label to each street of the route. On each street, we determine a reference point and connect the label and its reference point by a vertical line segment, the *leader*, whose length we can vary at runtime. In order to avoid jumping labels, we allow label–label overlaps but try to keep their area small. On the other hand, we try to keep the leaders at a desired length. We introduce a force-directed algorithm that manipulates the lengths of the leaders: labels repel each other and the reference points repel or attract their labels. Compared to an algorithm that does not vary the lengths of the leaders, our algorithm drastically reduces the area of label–label overlaps whereas the computation time does not change significantly.

Chapter 1

Introduction

We are continuously faced with a variety of annotations. Just have a look around. How many objects with *labels* do you see? Possibly, you see *textual* and *pictorial* labels. Some labels are extremely important. Would it not be strange looking at a bottle filled with a transparent liquid without any label that tells us something about its content? If you have not filled the bottle yourself, what is in there? Is it water? Is it alcohol? Or is it a toxic liquid? Other labels provide information that is just nice to have (for example, digits at analog wall clocks); some labels are even superfluous (for example, various logos).

In cartography, labels are a necessary tool to convey additional information on maps. If we, for instance, see only the boundary of a country, in many cases, we cannot identify the country correctly.

Map Design. The word *cartography* comes from the Greek χάρτης ("khartēs") meaning thick paper and γράφειυ ("graphein") meaning to draw. Cartography is the art and science of making maps. A map is a schematic representation of (a part of) the real world. Examples of maps are street maps, city maps, and thematic maps. As the displaying medium usually is much smaller than the represented part, maps are abstracted and size-reduced. There are three types of geometric shapes that help to represent the abstracted world: points represent *point features*, for example, (summits of) mountains or *points of interests*; examples for points of interest (or, *POIs* for short) are restaurants, gas stations, or hospitals. Polygonal lines represent *line features*, for example, streets or rivers. Polygons represent *area features*, for example, lakes or countries.

It is a natural goal that maps should be well comprehensible. For this reason, the degree of abstraction of the real world depends on the map *scale*. A map is in a *small scale* if the features shown on the map are rather small. Conversely, on *large-scale* maps, features are relatively large. The larger the scale, the more details can be shown. The smaller the scale, the more the map must be *generalized*. For example, in large-scale maps, each single house is shown as one rectangle; in small-scale maps, several houses are *aggregated* to one rectangle. In large-scale maps, all point features are displayed; the smaller the map scale gets, the fewer point features are shown, or *selected*. A polygon representing the boundary of a country in a map with a rather large scale has more edges than a map in smaller scale; the polygon is *simplified*. According to Beard and Mackaness [BM91], there are eight so-called generalization operators; we have mentioned three of them above: aggregation, selection, and simplification.

On the other hand, maps are typically augmented with additional information by, for example, colors, symbols, and, in particular, labels.

Static Label Placement. Good label placement is important to ease the map user's orientation. There are two aspects that a *good* labeling has to fulfill. First, there are some aesthetic aspects. Second, there should be an adequate number of labels such that the map coveys a suitable amount of information and each label is legible, that is, no two labels overlap. Aesthetic aspects for paper, or *static*, maps, have first been considered by the French cartographer Alinhac [Ali63] or the Swiss cartographer Imhof [Imh75]. Simultaneously but independently, they presented examples and rules for good and poor labeling. (At this point, we go without the rules; instead we mention appropriate rules in the corresponding chapters.) They gave, however, only common expert knowledge without any prove of quality. Their work was the starting shot for cartography becoming science.

It seems to be easy to place labels on maps but indeed professional cartographers (manually) place only 20 to 30 labels per hour on paper maps [CJ90]. In order to speed up the production of maps, in the 1980's, first steps towards automated label placement were made. Most work in the 1980's and 1990's examined the problem of attaching axis-parallel *rectangles* (representing labels) to point features in static maps. Many different *objective functions* have been considered. The general labeling problem can be formulated as follows: *Given a set of points in the plane, find a labeling such that no two labels overlap and the number of placed labels is maximized.* This problem is related to the packing problem where a given container (for example, a rectangle) is intended to be packed (that is, filled) with objects from a given set (for example, further rectangles) as densely as possible such that the objects neither overlap each other nor overlap the boundary of the container. The problems differ in that each label can only be placed within a certain area (within the container). Observe, however, that the above formulation for the labeling problem does not specify the relation of a label and the point feature it labels, its *reference point*. In general, there are two *labeling models*, the *fixed-position* model and the *slider* model [vKSW99]. In the fixed-position model, there is a discrete set of positions, or *candidates*, where a label can be placed. Quite common are models where a corner of the rectangular representation of the label must touch the reference point. The slider model permits an unbounded number of candidates. For example, the label can slide with its lower edge along its reference point. In some models, each reference point comes with a *weight* (or priority). The higher the weight, the more important it is to label the point. Then, the aim is to maximize the sum of the weights of the labeled reference points whereas we still require that no two labels overlap. Independently of the labeling model, maximizing the number of labels or maximizing the sum of the weights in an overlap-free labeling is NP-hard [FPT81, MS91, PSS$^+$03], that is, there are no algorithms that optimally solve the problems in suitable computation time [GJ79] (unless $\mathrm{P} = \mathrm{NP}$).

Yet, from the view point of geographic information science, that is, for the use case, maximizing the number of placed labels is not always desirable as labels can occlude important parts of the map or, more generally, they occlude too much of the map. In these cases, labelings could, for example, be thinned out by generalization approaches.

In other labeling-problem definitions that have been considered for the static case, all rectangular labels are placed. The aim was either to maximize the number of overlap-free labels [dBG12] or to maximize the font size such that the labeling is overlap-free [FW91]. In general, both problems are NP-hard, too. Further variants of the labeling problem vary the shape of the labels, for example, using squares or disk-shaped labels. Despite the NP-hardness of most variants of the labeling problem, several exact algorithms have been presented. Further, there are many algorithms that compute non-optimal solutions in suitable computation time. As the number of proposed algorithms and variants is large, we just point to the map labeling bibliography of Wolff and Strijk [1.1]. We conclude that, for practical purposes, the problem of labeling point features on static maps can be considered solved.

Interactive Maps. Nowadays, we rather use digital maps. Typically, these maps are *interactive*, that is, the user can manipulate the currently visible part of the map, the *view*, by *panning*, *zooming*, and *rotation operations*. Whereas static maps only provide one scale, we benefit from interactive maps as they can provide the entire world map at an unlimited number of scales. An example for such maps are Google Maps[1.2]. Some modern maps even allow for a *perspective view* (also *three-dimensional* or *3D view*), that is, the user can *tilt* the view from a *top view* (also *two-dimensional* or *2D view*) into a bird view and back. Examples for such maps can be found in the online map service Google Earth[1.3] and in *navigational devices* such as navigation systems or smartphones. For the sake of simplicity, in the remainder of this thesis, we refer to the *user*, meaning either the map user who can manipulate the view manually or a navigational device that can manipulate the view automatically.

Currently, interactive *3D virtual environments*, that is, maps with a perspective view that also allow for 3D objects (such as buildings), are getting more and more popular.

Dynamic Label Placement. Some older digital maps only permit discrete interactions. The reason is that the map indeed consists of many maps, one for each scale in a predefined set of scales, that is, the map is stored in *levels of detail* (*LoDs* for short). Each of these maps is generalized and possibly also labeled in advance. When the user zooms, the view shows the next coarser LoD or a mixture of the next smaller and the next larger scale. A problem with such approaches is, however, that the content of the view changes abruptly at certain scales and the user might lose context. On that account, Been et al. [BDY06] introduced the notion of *consistency* when placing labels in interactive maps:

[1.1] `http://i11www.iti.uni-karlsruhe.de/~awolff/map-labeling/bibliography`, accessed Feb. 9, 2015
[1.2] `https://maps.google.com`, accessed Oct. 1, 2014
[1.3] `https://earth.google.com`, accessed Oct. 1, 2014

they require that labels neither *flicker* nor *jump*. Flickering means that labels frequently disappear and reappear during a zooming operation; jumping means that, if a label must be placed at a new position, it moves smoothly to this position instead of suddenly vanishing and reappearing.

In modern maps, the placement of labels is dynamic; for example, if a street label leaves the view, the label should be placed such that it is visible again. Sometimes it is also desirable to prohibit labels to intersect the *view boundary*. In order to save computation time, for some digital maps a *tile-based* approach is used, that is, the map is subdivided into a grid. Labeling each cell individually reduces the search space for overlapping labels. Usually, in a tile-based approach, the labeling of several rows of tiles is precomputed and stored, that is, *cached*. If, however, a label overlaps two tiles, the placement of the two label pieces must be coordinated, otherwise the two pieces may not fit. A completely different approach is to use vector-based maps.

Adjusting the objective function of the static to the dynamic case is quite simple. We just require to maximize the number of placed labels in an overlap-free labeling over time under the constraint that the consistency criteria are satisfied. We observe that optimizing the number of placed labels in a static map is a special case of optimizing this number in a digital map. We obtain the static out of the dynamic case when assuming that the user does not interact with the map while the map is in the smallest scale that is possible. Consequently, the problem of maximizing the number of placed labels is NP-hard in the dynamic case, too.

Label Placement in Practice. In practice, interactive maps are challenging in that they have to react to user interactions in *real time*, that is, users can continuously manipulate the view without noticing the computation time that is needed for the update of the visible part of the map and the labeling. In some old maps, interactions are interrupted by the system. Then, users have to wait a perceivable amount of time until the map is updated. Only then, users can continue their interaction which is soon interrupted again. We deduce that placing labels dynamically requires very fast, that is, highly *efficient*, algorithms. Yet, maximizing the number of placed labels in an interactive map, while requiring that the labeling is overlap-free, is NP-hard. This makes the usage of *heuristics* or *approximation algorithms* inevitable. Heuristics efficiently compute non-optimal, but hopefully good, solutions. Approximation algorithms are efficient but also provably good, that is, we can show that, for any instance, their solution differs by at most a certain factor from an optimum solution. Unfortunately, sometimes even a quadratic running time does not satisfy the time requirements of interactive maps.

In computer graphics, the efficiency of an algorithm is typically expressed by the *frame rate*. When displaying a digital map, the content of the screen is drawn repeatedly; the content between two updates is a *frame*. For animation films, frame rates of 24 *frames per second* (or, *FPS* for short) are characteristic.

Our Contribution. In this thesis, we focus on dynamically placing labels in interactive maps that provide a 2D and/or a 3D view. We allow for continuously manipulating the view by (a subset of) the interaction types panning, zooming, rotating, and tilting. We use vector-based maps and do not cache labels. We mainly present heuristics for labeling point and line features. We aim for placing as many labels as possible in an overlap-free labeling while satisfying the consistency criteria, that is, labels neither jump nor flicker.

In order to verify that our algorithms are fast enough, we implemented them and measured the resulting frame rates. Indeed, all of our algorithms update the labeling in real time while directly reacting to user interactions; the user does not notice the computation time. We are of the opinion that our algorithms compute aesthetic and useful labelings. It remains to verify this with the help of user studies.

A typical question that arises when labeling interactive maps is which labels should be displayed on which scale. For example, if the map shows a country, labeling each single bakery of the country seems not to be appropriate (at least in most cases). We emphasize that we study *optimization problems* from the view point of *algorithmics*. Consequently, we always try place many labels. We do not propose any algorithms to thin out the computed labelings afterwards. Additionally, we assume that the given sets of features fit to the considered map scale. Yet, all but one of our algorithms can even deal with dynamically changing sets of features.

Concerning related work, most research about labeling problems has focussed on static maps; mostly on point-feature labeling. Only few papers deal with line and area-feature labeling. Although interactive maps are widely used, there is only little research about this topic. Additionally, most of the presented algorithms for labeling interactive maps have their weaknesses; for instance, the labelings are not consistent. With this work, we intended to close the gap between theory and practice; between papers dealing with static labeling problems and common navigational devices.

Outline of the Thesis

We start this thesis with a chapter that asks what users consider a *good* labeling. The remainder of the thesis is organized in two parts, both dealing with algorithms for labeling interactive maps: Part I is about *labeling point features*; Part II is about *labeling line features*. We have not developed algorithms for labeling area features in interactive maps or placing labels in interactive 3D virtual environments (where also the surrounding, for instance, buildings or the terrain, can occlude labels).

Chapter 2: A Preliminary User Study

Chapter 2 is devoted to user studies. Possibly, we are all familiar with digital maps. What, however, makes a map aesthetic? What makes it useful? In order to answer these questions, we started our research project about labeling interactive maps by going through existing user studies for paper and interactive maps. As these studies could not answer all of our questions, we conducted a small preliminary study ourselves.

As we had not prototypes of labeling algorithms for digital maps at that time, we were forced to base our study on static pictures. We asked the participants about the aesthetics and usefulness of several pictures with labelings under the assumption that these labelings would be shown in navigation systems. We came to the conclusion that, in a 3D view, labels should adjust to the perspective and that the concept of using two different labeling styles for streets on a user's route and the remaining streets is worth trying. We were aware of the fact that a similar study based on moving pictures can change the results. Nevertheless, we took the results as an initial guideline.

This chapter contains parts of joint work with Benjamin Morgan, Jan-Henrik Haunert, and Alexander Wolff [SMHW15].

Part I: Labeling Point Features in Interactive Maps

In the first part of this thesis, we study the problem of labeling point features. We first consider the *offline*-labeling problem of placing disk-shaped labels; offline means that the labeling is computed in a preprocessing (note that this does not mean that the labeling is static at runtime). In addition, we introduce a dynamic algorithm using a slider model: we permit each label to move along its corresponding reference point in order to make space for other labels.

Chapter 3: Optimizing Active Ranges for Labeling Point Features

In Chapter 3, we consider the problem of annotating large point sets with disk-shaped labels in maps where the user can continuously pan, zoom, and rotate. Such disk-shaped label represent, for example, pictorial labels that are placed to POIs. Due to the shape of the labels, we only have to update the labeling while the user zooms. We require that, while the user zooms, labels maintain constant size on the screen. Further, labels might not flicker: while the user zooms out, a label that vanishes is not allowed to appear again; if the user zooms in, the label appears at the same scale $1 : z$ as it vanished. Consequently, each label ℓ is placed in a range $(0, z(\ell)]$ of *scale factors*. We aim for an overlap-free labeling that maximizes the value of the objective function, which we define to be the sum over the upper bounds of the ranges of all labels. This problem is NP-hard [BNPW10].

The problem directly corresponds to the generalization problem of selecting points which we also introduce briefly.

We present a tool that solves our problem optimally but has a very high computation time in practice. For this reason, we also give two variants of a new heuristic. Our approaches build a data structure in a processing. We obtain the labeling at runtime by just querying the data. With this, our approaches are real-time capable. In order to verify that our heuristics compute quite good solutions, we have also implemented our approaches. We compare the results in terms of the values of the objective function as well as the computation time. Further, we compare our results to a similar heuristic that have been introduced by Been et al. [BNPW10]. We conclude that, if the number of placed labels over scales matters, it is reasonable to apply our heuristic.

Figure 1.1 further motivates our problem. If the user zooms out, the reference points get closer. Maintaining constant size on the screen, labels might overlap. On that account, we need to select the labels to be displayed. In order to include the information that not all labels are displayed, each label ℓ could additionally display a number that indicates the number of labels that could not be placed because of ℓ. In our opinion, most current digital maps support only a rather poor labeling of POIs. In an unknown city, it is often hard to quickly find a place to eat or the shopping mile by means of the interactive map. Only after we have zoomed to a quite large scale, POIs get visible. The new scale, however, is such large that we lost the context of the city as a whole. Although we suppose that showing POIs only at large scales is on purpose, we think that this is not very useful.

This chapter is based on joint work with Dennis Allerkamp, Jan-Henrik Haunert, and Alexander Wolff [SAHW13].

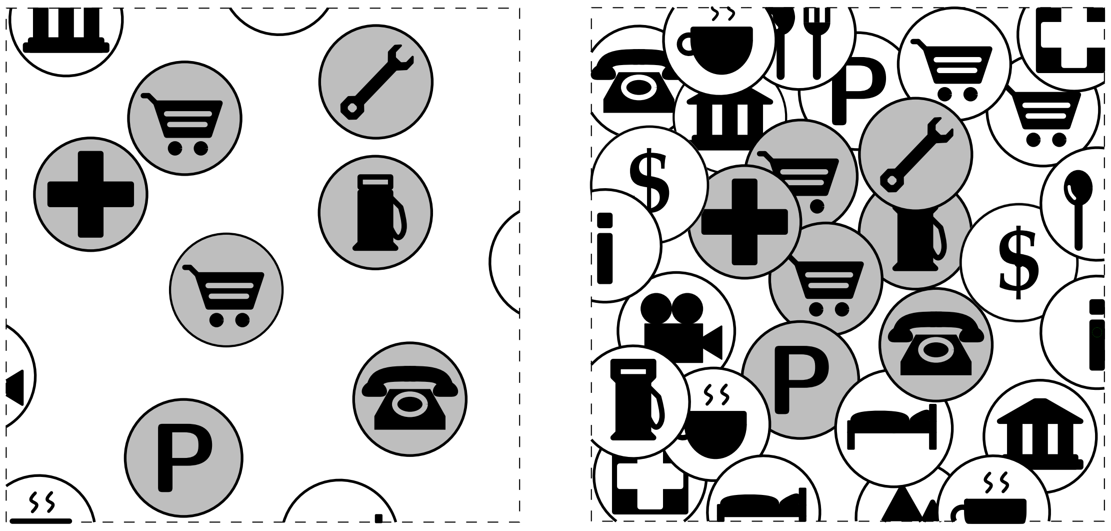

(a) initial situation (b) after zooming out

Figure 1.1: We require that each label has constant size on the screen. Thus, if the user zooms out, labels might overlap. We marked labels that were completely visible before zooming out by a gray background.

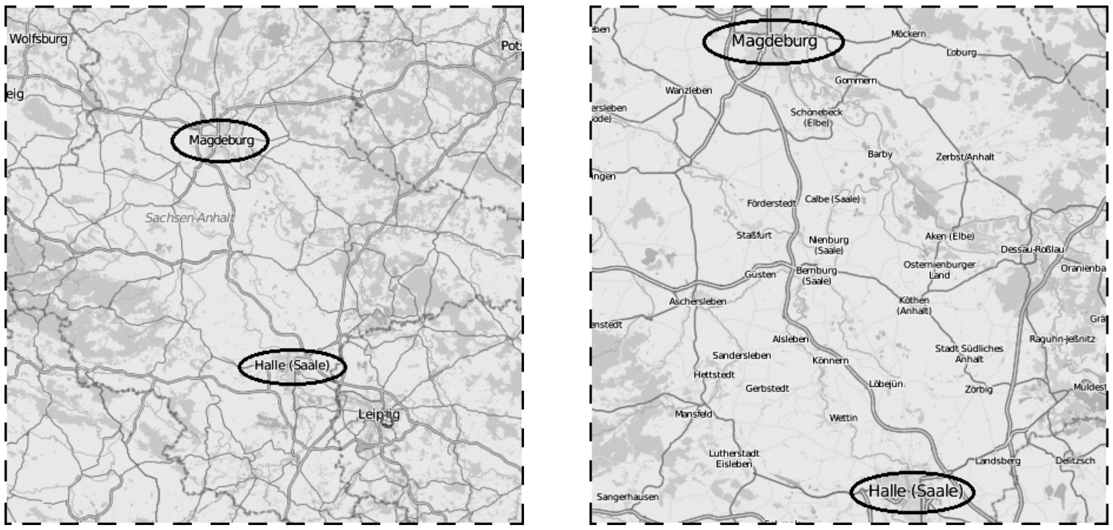
Chapter 4: Labeling Point Features with Sliding Labels

In Chapter 4, we investigate the problem of labeling point features in interactive maps where the user can continuously pan and zoom. We allow the labels to slide with their bottom edges along the points they label. We assume that each point comes with a weight. Given a dynamic scenario with user interactions, our objective is to maintain an overlap-free labeling such that, on average over time, the sum of the weights of the labeled points is maximized. Even the static version of the problem is known to be NP-hard [FPT81].

We present an efficient and effective heuristic that dynamically labels points with sliding labels on runtime and in real time. Our heuristic proceeds incrementally; it tries to insert one label at a time, possibly pushing away labels that have already been placed. To quickly predict which labels have to be pushed away, we use a geometric data structure that partitions the view. With this data structure we were able to double the frame rate when rendering maps with many labels.

Many of the current digital map products are not satisfactory in terms of label placement; they block large areas around labels in order to avoid that labels overlap when the user manipulates the view; see Figure 1.2.

This chapter is based on joint work with Jan-Henrik Haunert, Alexander Wolff, and Dennis Zwiebler [SHWZ14].



(a) initial scale; sparse labeling (b) when zooming in, several labels appear

Figure 1.2: Extracts from OpenStreetMap[1.4]. The map on the left has a sparse labeling, although there are many cities that could be labeled (see map on the right). Obviously, there is enough space to label (at least some of) them.

Part II: Labeling Line Features in Interactive Maps

The second part of this thesis is devoted to the problem of labeling line features, or, more precisely, the problem of labeling streets. On the one hand, we consider a problem where labels look as painted on the streets; on the other hand, we study the case of placing *billboards*, that is, we place axis-parallel rectangular labels with some distance to their

[1.4] `http://www.openstreetmap.de/`, accessed Oct. 31, 2013 corresponding streets whereas we maintain the label–object association by attaching a *leader*, say for example, a line segment, that connects the label with its street.

Chapter 5: Labeling Streets with Embedded Labels

In Chapter 5, we study the problem of dynamically labeling line features in interactive maps where the user can continuously pan, zoom, rotate, and tilt. More precisely, we label streets that we visualize with spatial extent. Our labels are *embedded* into their corresponding streets, that is, labels follow the curvature of the streets, they do not move with respect to other map objects, but they scale in order to maintain constant size on the screen. In a 3D view, labels can be perspective. To the best of our knowledge, this is the first work that deals with curved labels in interactive maps.

Our objective is to label as many different streets as possible and to select nice-looking label positions while forbidding labels to overlap at street junctions. We present a simple but effective algorithm that takes curvature and junctions into account and produces aesthetic labelings. Averaged over all interaction types, our implementation runs with more than 85 FPS.

Although most digital map services support labeling streets, in our opinion, their solutions are not completely satisfactory. For example, while the user pans, labels can leave the view. In order to avoid unlabeled streets, some services label the same street several times. This is surely useful but, in many cases, label clusters harm the aesthetics of the map. Even when permitting clusters, sometimes, a street has no label within the view (at least at small displays such as used for smartphones); see Figure 1.3. Another problem of some of these services is that, while zooming, some labels jump to completely new positions.

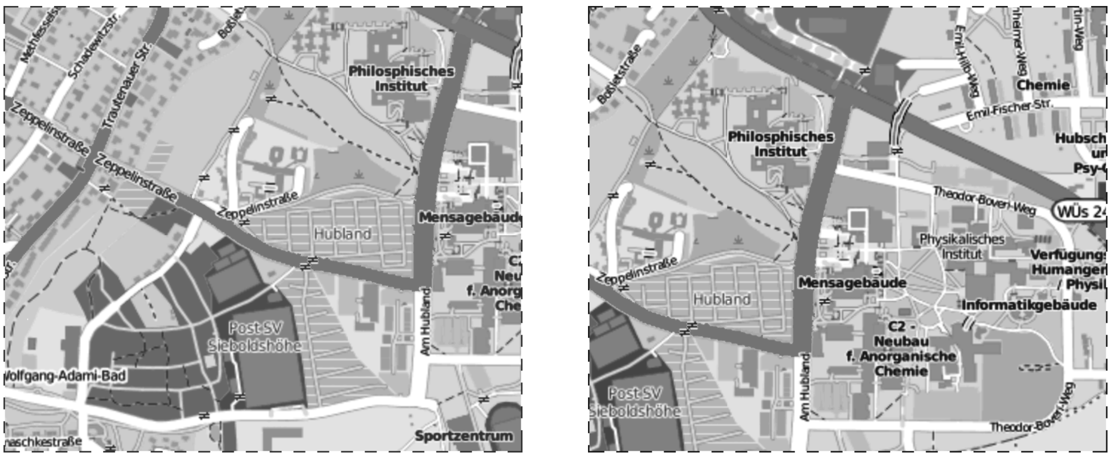


Figure 1.3: Extracts from OpenStreetMap[1.5]. From left to right: If we move the visible part of the map to the right, we do not know the name of the gray street ("Zeppelinstraße") any longer.

[1.5] `http://www.openstreetmap.de/`, accessed Dec. 29, 2014

This chapter is based on joint work with Alexander Wolff and Jan-Henrik Haunert [SWH14].

Chapter 6: Labeling Streets Along a Route with Billboards

In Chapter 6, we again consider the problem of labeling line features in interactive maps where the user can continuously pan, zoom, rotate, and tilt a perspective view of the scene. To be exact, we attach billboards to streets that belong to a user's route whereas we assume that the future course of the route within the currently visible part of the map is known or well predicted. Our leaders are vertical line segments.

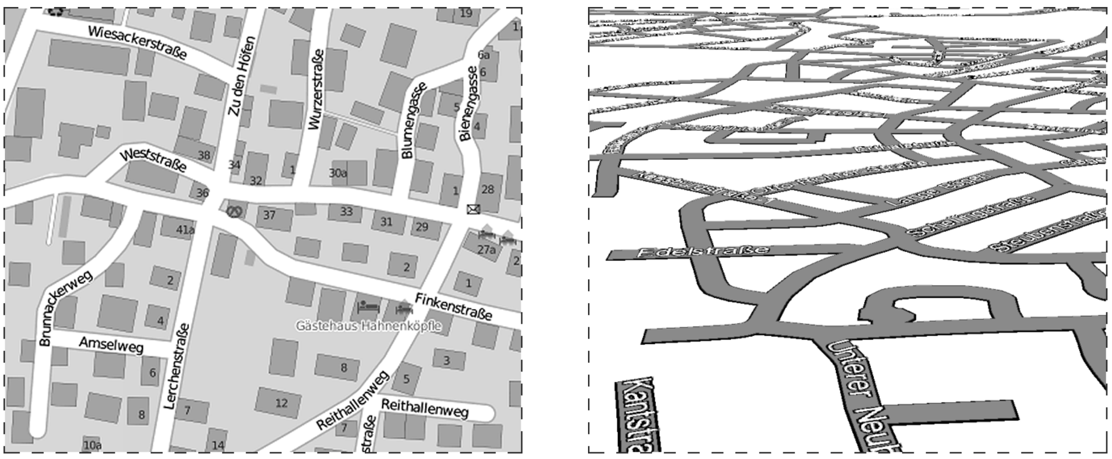
Our goal is to maintain an overlap-free labeling that reacts to changes of the view in real time. To this end, we dynamically vary the lengths of the leaders. In order to achieve that labels move smoothly, we do not strictly forbid label–label overlaps. We present a force-directed algorithm that applies forces to the leaders. By the change of the leader lengths, the forces make overlapping labels repel each other while they also keep leaders as close as possible to a desired length. On real-world data with a realistic number of labels, we obtain frame rates of more than 420 FPS. Compared to an algorithm permitting only a fixed leader length, we drastically reduce the total overlapped area (induced by label–label overlaps): the overlap caused by our algorithm is less than 2% of the overlap caused by the algorithm with a fixed leader length.

We observe that embedded labels as in Chapter 5 sometimes need a high reading time as they are hard to decipher, for example, if the part of the street at which the label is placed is quite curvy or, in the case that labels are subject to perspective distortion, the label is the worse decipherable, the more the view is tilted; see Figure 1.4. A high reading time is, of course, unfavorable for navigation systems. In order to make the streets that are most important for the user well and thus faster legible, we decided to place billboards which are axis-parallel.

Chapter 6 is based on joint work with Benjamin Morgan, Jan-Henrik Haunert, and Alexander Wolff [SMHW15].

Chapter 7: Combination of Two Street-Labeling Algorithms: Embedded Street and Billboard Labeling

The final chapter, Chapter 7, is also devoted to the problem of labeling streets in interactive maps where the user can continuously pan, zoom, rotate, and tilt. More precisely, we attach billboards to streets on routes that lead users to their targets and we attach embedded labels to the remaining streets. As before, for the labeling using embedded labels, we aim for an overlap-free and aesthetical labeling; for the labeling using billboards, we require that labels overlap as few as possible whereas the label movement for solving overlaps should be smooth. We accept billboards overlapping embedded labels. All in all, this results in an aesthetic labeling which transfers much information and improves the legibility of the information that is most important for the user of a navigational device.



(a) the reading time for curvy streets is higher, for example, for "Bienengasse" (top right) or "Brunnackerweg" (bottom left)

(b) if labels are perspective, the more the view is tilted, the more difficult it is to decipher the labels

Figure 1.4: Extract from OpenStreetMap[1.6](left) and our own implementation of the algorithm of Chapter 5 (right). Some embedded labels need a higher reading time.

In order to compute such a labeling, we consider our two street labeling algorithms of Chapter 5 and 6 a second time. We have combined the code for dynamically placing embedded labels and dynamically varying the length of leaders of billboards along a route. On real-world data, the combining algorithm obtains average frames rates of more than 90 FPS. As we have already introduced the concepts of the algorithms for labeling streets with embedded labels and with billboards in Chapter 5 and 6, respectively, in this chapter, we only present some tests and figures of the algorithm combining the two concepts.

The motivation for combining the algorithms for placing embedded labels and placing billboards is that the algorithm for placing billboards is not completely satisfactory with regard to user orientation. If the street network is rather uniformly and dense, users might not know at which junctions they have to turn; see Figure 1.5. When using a navigation system, we are usually tracked by GPS. Since the tracking sometimes is inaccurate, further information is helpful. Under the assumption that streets in the real world have plates, streets that do not lie at the route are helpful if they are labeled in the digital map.

We are not aware of any navigation system or research work that supports such a labeling. On that account, we submitted the idea as invention disclosure [NSW12]. As of this writing, the submission was neither accepted nor rejected.

[1.6]`http://www.openstreetmap.de/`, accessed Jan. 28, 2015

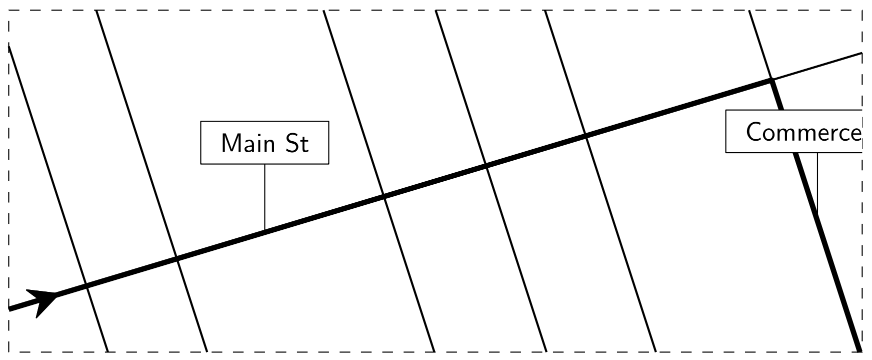


Figure 1.5: For users of navigational devices, it might be hard to say at which (real-world) junction they have to turn as they have no information about the streets near the junction. The triangle (bottom left) represents the user, the thicker line indicates a route that leads the user to a target.

Chapter 2

A Preliminary User Study

When we started our research project about labeling interactive maps, we first collected several ideas of how a good labeling could look like. It is, however, quite subjective whether a labeling is aesthetic or useful. For some but not all of our ideas, we could find the solutions in previous user studies. In order to answer the remaining questions, we ourselves conducted a preliminary user study with printed, steady pictures before developing algorithms and implementing them. We are fully aware of the fact that the results of this study can differ a lot compared to a study on interactive maps. We emphasize that the study should only give a hint which ideas could be worth trying.

In the remainder of this chapter, we first give a short overview over related user studies (see Section 2.1). Then, we present and discuss our study about point-feature labeling (see Section 2.2). We examined which factors should influence the size of a label in a 3D view. In paper maps, the size of a city label usually indicates the size of a city. Should this concept be maintained for 3D views? Next, we present and discuss our study about line-feature labeling (see Section 2.3). We consider the question whether the concept of using two different labeling styles for streets that belong to a user's route and the remaining streets is worth trying. Moreover, we ask whether labels in a 3D view should adjust to the perspective. We finally conclude that the size of a city should not influence the size of a label in a 3D view, the concept of two different labeling styles is worth trying, and labels should be subject to the perspective. After implementing the algorithms, our preliminary results should be verified in a final user study supporting moving pictures for deciding which types of labelings should be supported in interactive maps.

2.1 Previous and Related Work

A fundamental problem that preceeds the design of labeling algorithms is to define what a *good* labeling is. Sometimes rules are accepted because they are given by approved experts, such as Imhof [Imh75] or Alinhac [Ali63]. Another possibility is to analyze common practice. Strijk [Str01], for example, surveys common ways to label line objects; he uses the results of his survey when designing his labeling algorithms. The last approach, we point out, is to conduct comprehensive user studies.

Tinker [Tin72], Koriat and Norman [KN85], and Wigdor and Balakrishnan [WB05] investigate the correlation of the reading time and the orientation of the text. They all come to similar conclusions in their user studies. If we say that horizontally written text has a rotation angle of 0°, the user studies observe that rotations of up to $\pm 60^\circ$ have almost no effect on the reading time; rotations between $\pm 60^\circ$ and $\pm 120^\circ$ increase the reading time more and more. The longer the words, the longer the reading time. Rotations between $\pm 120^\circ$ and $\pm 180^\circ$ evoke the maximum reading time. Within this interval, however, the reading time stays constant. With these findings, the authors confirm Imhof's rule [Imh75] dealing with the orientation of the text: text should be written as horizontal as possible.

Larson et al. [LvDCR00] extend these results by 3D rotations. They rotate text sideways around a vertical axis (that is, into the third dimension) and examine legibility of text. The authors deduce from their user study that the legibility of labels with a high quality and large font size are almost unaffected by this 3D rotation. The smaller the font size, the more legibility decreases. Due to perspective distortion, text rotated to the right (clockwise) is slightly better legible than text rotated to the left.

Harrison and Vicente [HV96] consider the legibility of text written on semi-transparent rectangles that are drawn on different backgrounds. By means of a user study they show that rectangles with a transparency between opaque and half-transparent are almost equally-well legible. For us this means that we can use semi-transparent label boxes in order to uncover more details of the map in the background while diminishing the legibility only slightly.

The authors additionally consider the improvement of the reading time when outlining the characters of the text. They conclude that an outline only improves the legibility if the transparency of the rectangle is between half and fully-transparent. Several participants claim, however, that they do not like the outline but rather prefer half-transparency (or less). For labeling line features with embedded labels, we currently outline the text. This is, however, configurable.

As Harrison and Vicente [HV96], Jankowski et al. [JSI$^+$10] approve in their user study that text placed on a semi-transparent rectangle is better legible than text with an outline (without a semi-transparent rectangle behind) when placing text in videos or on 3D graphics, that is, they partially use moving pictures. Further the study shows that white text on a semi-transparent black rectangle is as well readable as black text on a semi-transparent white rectangle.

Also concerning moving pictures, Ooms et al. [ODF09] hypothesize that a label that abruptly changes its position due to a change of the currently visible part of the map distracts the user. The authors suggest conducting a user study in order to learn whether this hypothesis holds. As of today, no answer has been given.

Maass et al. [MJD07b] analyze new challenges that emerge in interactive 3D virtual environments. First, the authors deal with the problem that 3D objects can occlude labels. They suggest either using transparency for the occluding object or, in the case of billboards, using dotted leaders in order to give a hint that the reference point is hidden. They also state that it is acceptable to draw a leader over objects if the user profits from the damage of the depth cues. Second, the authors point out that indicating the number of inhabitants of a town by the size of a label, as common in static maps, cannot be realized in perspective views. The size of the labels should rather be equal or shrink with the distance to the user. With a constant or even shrinking label size, we can place more labels in the background. Besides, the authors observe that, in a 3D view, labels can be rendered with and without perspective distortion. They admit that there is only little known about the impact and usefulness of these two rendering possibilities. Finally, the authors advise against using shadows in the case that shadows might irritate the user.

Bachfischer [Bac05] deduces, based on findings of other works [PT46, Reh81], that, if the lower part of a word or text is occluded, it is better legible than in the case that the upper part of the text is occluded. This gives us a hint that overlaps in a 3D view where labels are overlapped from the bottom up (under the assumption that depth cues are correct), might only little or not all disturb the user.

Further, Maass et al. [MJD07a] present the results of a user study dealing with labelings in interactive 3D virtual environments using billboards. The authors examine the problem of leaders inducing wrong depth cues. This happens, for instance, if a leader whose reference point obviously lies behind a building is drawn over the building. Most of the participants of the user study like correct depth cues better than wrong depth cues. The authors finally suggest introducing a parameter that measures the perceivable perspective disturbance. If the parameter is applied in labeling algorithms, it sometimes permits wrong depth cues. Concerning leaders, our approach for labeling streets along a route with billboards (see Chapter 6) maintains correct depth cues.

Vaaraniemi et al. [VTW12] describe the results of an expert study. Their first conclusion is that labels in a 3D view should shrink with distance to the user in order to create a better understanding of the depth (this approves the analysis of Maass et al. [MJD07b]). We consider this result in the implementation of our algorithms that label maps in a 3D view (see Part II).

Moreover, Vaaraniemi et al. ask the experts whether streets should be labeled embedded (that is, whether a label should follow the curvature of its street), by horizontally-written billboards with leaders, or aligned to a straight line that has a similar rotation as the part of the street at which the label is placed. Four of six experts judge billboards as very legible, although they also note that a high search time is required to associate a label with its object. On the other hand, five of six experts like the embedded labels, but also point out that if the part of the street where the label is placed has a strong bend, an embedded label might be hard to decipher. Three of six experts observe that embedded labels yield a good label–object association.

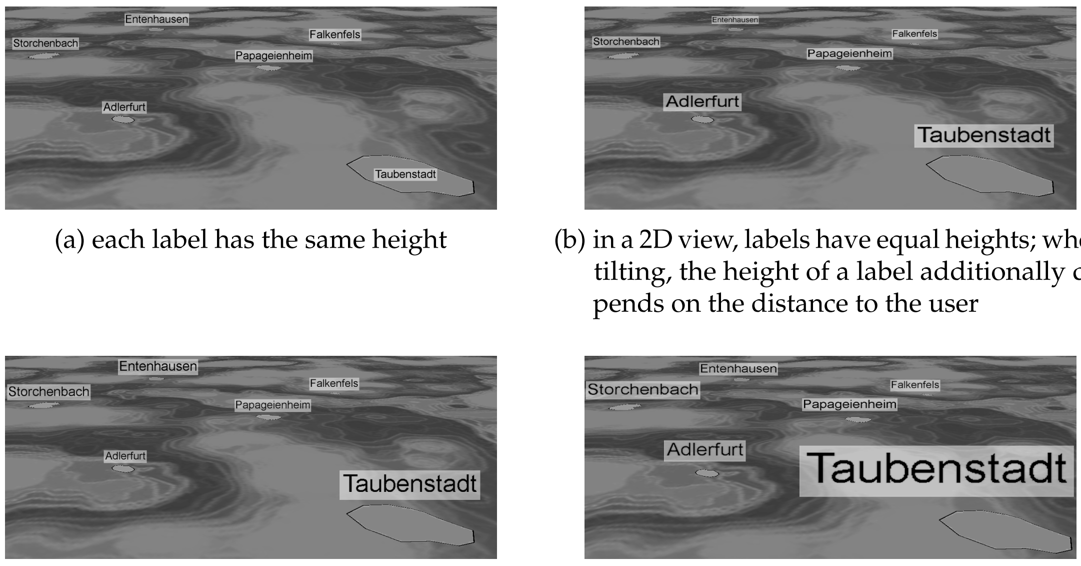
Based on the above-mentioned findings, we propose the compromise of using horizontally-written labels if the legibility of the label is very important and labeling the remaining streets embedded. More precisely, concerning navigational devices, for labeling streets on roads that lead users to their destinations, we use billboards (see Chapter 6). In order to improve the label–object association, we suggest attaching vertical leaders that anchor the label at the center of the label's bottom edge. As all other streets are less important for the users, we label them with embedded labels (see Chapter 5).

2.2 Labeling Point Features

In the first part of our study, we considered the problem of labeling point features in a 3D view with *regularly-rendered* labels, that is, with labels that have no perspective distortion. We examined which factors should influence the size of a label.

2.2.1 Experimental Setup

Maass and Döllner [MJD07b] listed three common ways to determine the size of a label of a city. For each of these possibilities, we designed a figure (see Figures 2.1(a)–2.1(c)) and complemented these three possibilities by another solution (see Figure 2.1(d)). We posed two questions and set one task to the participants. (i) The participants had to decide which of the four figures they liked best in terms of aesthetics. (ii) The participants should rate how important they consider it to be able to determine the number of inhabitants of a city by the size of a label. (iii) The participants should describe which factors influenced the sizes of the labels. Finally, every participant filled in a personal questionnaire about age, gender, subject of study, and the previous use of navigational devices.



(a) each label has the same height

(b) in a 2D view, labels have equal heights; when tilting, the height of a label additionally depends on the distance to the user

(c) the height of a label depends on the number of inhabitants of the city

(d) in a 2D view, the height of a label depends on the number of inhabitants of the city; when tilting, the height of a label additionally depends on the distance to the user

Figure 2.1: Figures we showed to the participants of our preliminary study.

2.2.2 Procedure

We arranged the four figures on a single sheet of A4 paper in landscape format. In order to avoid that the answer to the task influences the answer to the two questions, we printed the questions and the task on different sheets. We also printed the demand only to read the task after answering the questions. We did not, however, control if the participants complied the request. To each participant, we handed out one copy of the figures, the questions, the task, and the personal questionnaire. We did not limit the process time.

2.2.3 Participants

In total, 19 voluntary university students participated in our preliminary user study. They were aged between 19 and 25, with an average age of 21.8 years. One participant was female, 17 participants were male, one did not answer this question. Every student studied a technical subject. More precisely, 47% of the participants studied computer science, 47% studied mathematics, and 6% studied physics. Only 10% of the participants stated that they often use a navigational device. Nevertheless, 63% use navigational devices infrequently; the remainder never uses a navigational device. We remark that this study was in 2011. According to statista.com[2.1], in July 2011 18 million Germans used smartphones; until May 2014 this number raised to 41 million.

Finally, we asked the participants in which situations they use a navigational devices. (Multiple answers were possible.) As expected, most of them, namely 63%, replied that they use a navigational device while driving a car; 16% of the participants use a navigational device while walking, 5% while cycling.

2.2.4 Results and Discussion

We show the results of our preliminary study about labeling cities in a 3D view in Table 2.1 and discuss them in the following.

Table 2.1 shows that, in terms of aesthetics, the participants do not like the solution presented in Figure 2.1(d). The reason could be the striking label in the front. A figure with more similar font sizes could have yielded other results; on the other hand, the additional information about the number of inhabitants might get lost. Even provided that Figure 2.1(d) is rather poorly designed, we expect that a better example would not have changed our final decision.

About 47% of the participants judge the labeling in Figure 2.1(b) as the most aesthetic one. The next-best-rated labeling is that in Figure 2.1(c). It is preferred by 37% of the participants. This is no obvious result at all. If we yet consider the answers to the question which factors, in the opinion of the participants, influence the size of the labels, the solution in Figure 2.1(b) is the better one: 95% of the participants give a correct description for Figure 2.1(b) but only 53% for Figure 2.1(c) (whereas we rate a

[2.1] `http://de.statista.com/statistik/daten/studie/198959/umfrage/anzahl-der-smartphonenutzer-in-deutschland-seit-2010`, accessed Feb. 11, 2015

evaluation of the figures		
	most aesthetic	correct description
Fig. 2.1(a)	16%	95%
Fig. 2.1(b)	47%	95%
Fig. 2.1(c)	37%	53%
Fig. 2.1(d)	0%	21%

dependence on inhabitants	
not important	11%
little important	37%
important	42%
very important	5%
undecided	5%

Table 2.1: Results of our preliminary study about attaching regularly-rendered labels to point features in a 3D view and the question of how important it is to relate the size of a label with the number of inhabitants of the corresponding city.

missing answer as wrong answer). The table shows that, in general, the participants describe figures with equal label heights (Figure 2.1(a) and 2.1(b)) more often correctly and even no participant leaves these descriptions out. To the contrary, for each of the two labelings that depend on the number of inhabitants (Figure 2.1(c) and 2.1(d)) two answers are missing. Surprisingly, for Figure 2.1(c), many participants state that the size of a label depends on the number of inhabitants of the city and the distance to the user. In contrast, for Figure 2.1(d), many participants explain that the size of a label only depends on the number of inhabitants. That is, many participants mix up the factors that influence the two solutions presented in Figure 2.1(c) and 2.1(d). Recall that all the participants are young students with a technical background, that is, they usually have good spatial visualization abilities.

Concerning the wish to realize the number of inhabitants of a city by the size of its label, there is also no unique result. About 48% of the participants judge the relation as less or not at all important, 47% consider it important or even very important. We have to admit that the formulation of this question was inaccurate as we did not specify in which type of maps; static or dynamic. Nevertheless, in the end, this question does not influence our conclusion.

We conclude that, in digital maps that permit a 3D view, for the bird view, labels should have equal heights; when tilting, labels should shrink with distance to the user (see Figure 2.1(b)). We should use this style for labeling cities as well as for attaching billboards to streets. With these results, we approve the analysis of Maass et al. [MJD07b] and the expert study of Vaaraniemi et al. [VTW12]. Nevertheless, in paper maps in a 2D view, the size of a label for a city is typically related to the size or importance of the city. On that account, we suggest applying other stylistic elements for these characteristics, for example, we could vary the font or the color.

2.3 Labeling Streets

A navigational device usually shows a route that leads the user to a specified target. We observe that, in a 3D view, embedded street labels are sometimes hard to read. Simultaneously, the names of the streets that are contained in the route are essential for the user. We came up with the idea of attaching billboards to streets of the route and embedded labels to the remaining streets. With our study, we wanted to learn if this concept could be accepted by the users.

2.3.1 Experimental Setup

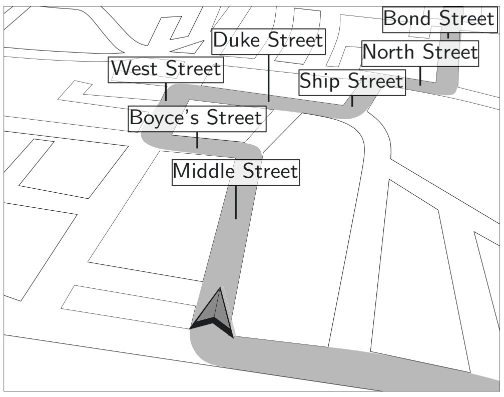
When labeling streets with billboards, there are many ways to place the label relative to its reference point. We designed four figures, each showing the same street network but with different locations of the labels and thus different directions of the leaders; see Figure 2.2. We posed the participants two questions. (i) Which of the four labelings is the most aesthetic one? (ii) Which of the four labelings is most useful with regard to navigational devices?

Another question for street labels concerned the way of rendering labels. Maass et al. [MJD07b] proposed two possibilities for rendering a label that follows the curvature of its street in interactive 3D virtual environments: an embedded label is rendered as if it were painted on its street, that is, it is subjected to the perspective; in contrast, a regularly-rendered label has no perspective distortion. We examined one task and one question. (i) In order to test which of the two styles is more useful for navigational devices, that is, which one is easier to decipher in a short time interval, we set the task to find typing errors in two almost equal labelings. For that purpose, we placed fictional street names to a synthetic map; see Figures 2.3 and 2.4. In the figure with typos, we marked one typo in advance to give the participants a hint on how to solve the exercise. We hided four further typos in each of the faulty labelings. (ii) Further, we asked the participants which of the two styles they preferred in terms of aesthetics.

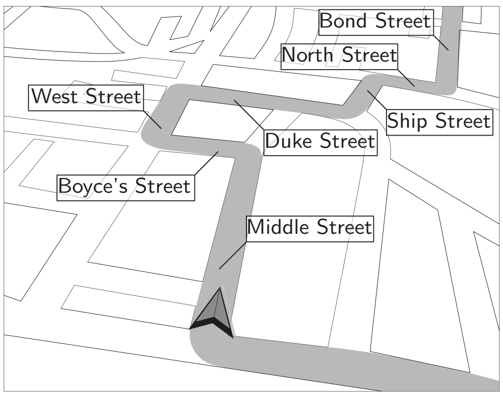
It is quite natural that embedded labels that scale with the distance to the user and lie in the upper part of the view usually are hard to decipher. On that account, we finally designed a figure showing a street network with embedded labels whereas we removed the labels in the upper third of the view (unfortunately, we do not have permission to publish the figure). We asked the participants if the missing labels in the upper part of the map troubles them.

2.3.2 Procedure

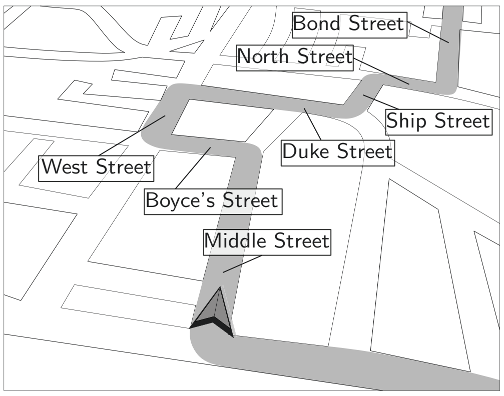
We designed a questionnaire with the above-mentioned questions and exercises. We divided the study into two parts. For the first part, we printed Figure 2.2 on a single sheet of A4 paper in landscape format and the figure with missing labels in the upper part of the view on half of a sheet of A4 paper in portrait format; we printed all the corresponding questions on another sheet. To each participant, we handed one copy of these printings. We did not limit the process time for this first part.



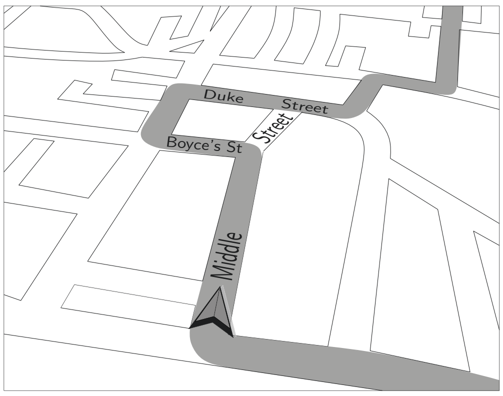
(a) vertical leaders



(b) leaders are rotated by $k \cdot 45°,\ k \in \mathbb{N}$

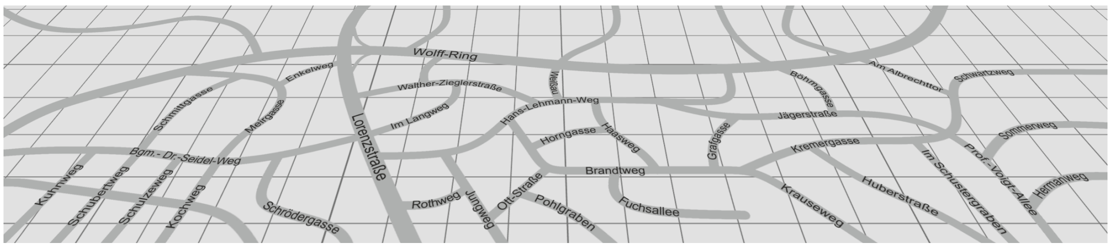


(c) arbitrarily rotated leaders

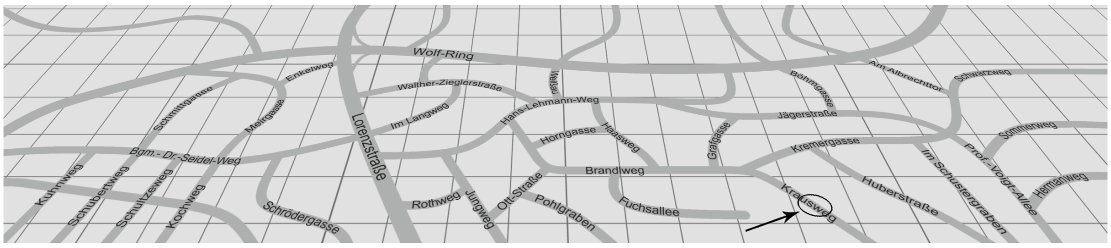


(d) embedded labels

Figure 2.2: Figures similar to those we have shown to the participants of our preliminary study. (We do not have permission to publish the original figures. They differ in that we removed further map objects and the embedded labels of streets that are not contained in the route.)

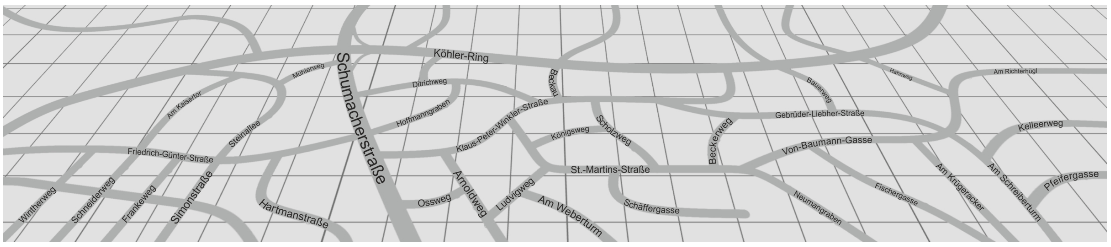


(a) "correct" labeling

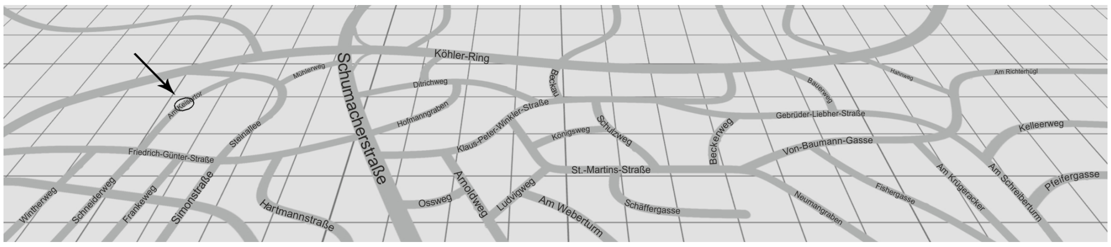


(b) labeling with typos; we exemplarily marked one error

Figure 2.3: Embedded labels look as they were painted on their streets. In our preliminary study, we asked the participants to find typos in the bottom labeling.



(a) "correct" labeling



(b) labeling with typos; we exemplarily marked one error

Figure 2.4: Regularly-rendered labels have no perspective distortion. In our preliminary study, we asked the participants to find typos in the bottom labeling.

For the second part, we printed the *search images*, that is, Figure 2.3 and Figure 2.4, on one sheet of A4 paper in landscape format each and the question on another sheet. In order to prevent that the results of the two exercises are distorted by learning effects, we changed the order of the exercises in half of the questionnaires. For answering the question, we did not limit the time; for finding typos in the search images, the participants had one minute each. We determined the process time of one minute by asking two colleagues to find the four differences in the two search images. Both colleagues started with the search image of Figure 2.4 (regular). One colleague needed 2:40 minutes for the image shown in Figure 2.4 (regular) and 3:21 minutes for the search image of Figure 2.3 (perspective); the other colleague needed 4:14 minutes for the image shown in Figure 2.4 (regular) and 1:20 minute for the search image of Figure 2.3 (perspective).

In order to ensure the time limit in the study, we handed out the copies of the second part separately and covered. By voice commando the participants turned the sheets. At the same time, we started a countdown which automatically rang after one minute. We did not communicate the total number of typos.

As the study about labeling point features and the study about labeling line features indeed were conducted in one, see Section 2.2.3 for information about the participants.

2.3.3 Results and Discussion

We show the results of our preliminary user study about labeling line features in a 3D view in Tables 2.2, 2.3, and 2.4 and discuss them in the following.

At first glance, when placing labels along a route, the participants consider embedded labels both more aesthetic and more useful; see Table 2.2. On closer inspection, there is no obvious result if we compare the results for using billboards and the result for embedded labels. Regarding aesthetics, 53% of the participants prefer billboards, whereas 47% prefer embedded labels. On the other hand, 43% think that billboards are more useful in navigational devices, 53% of the participants judge embedded labels to be more useful. Possibly, we should have split the questions. First, we should have asked the participants if they prefer one of labelings using billboards or the labeling using embedded labels. Every participant that prefers a labeling with billboards, must additionally select the preferred billboard labeling. Currently, among the three labelings using billboards, the participants favor the solution using vertical leaders.

We remark that using two different labelings styles for streets, that is, labeling streets along a route by billboards with leaders and the remaining streets embedded, is innovative; many people are yet suspicious of innovations. On that account, we conclude that labeling streets along a route with billboards is a concept that is worth trying. We should not forget that the study was conducted by means of static pictures and moving pictures possibly also influence the results.

Table 2.3 gives an overview of the typos the participants found. The group that started searching typos in the labeling in Figure 2.4 (regular) found only 32% of the errors in their first search process. In the labeling in Figure 2.3 (perspective), this group found

labels along a route			
		more aesthetic	more useful
billboards	Fig. 2.2(a)	32%	32%
	Fig. 2.2(b)	21%	11%
	Fig. 2.2(c)	0%	5%
	$\sum$	53%	48%
embedded	Fig. 2.2(d)	47%	53%

Table 2.2: Results for the question about the style of labels which are placed at streets contained in a route.

perspective labels (Figure 2.3)	
Wolf(f)-Ring	68%
Schul(t)zweg	58%
Schwar(t)zweg	58%
Brand[t/l]weg	53%
total found errors	59%
more aesthetic	79%

regular labels (Figure 2.4)	
Hof(f)manngraben	47%
Hartman(n)straße	32%
Fis(c)hergasse	53%
Sch[o/u]lzweg	42%
total found errors	43%
more aesthetic	26%

Table 2.3: Results for the task of finding typos. Letters in parentheses were left out in one of the labelings, letters in square brackets were exchanged. For every typo, we give the percentage of participants that found it. We additionally show the results concerning aesthetics.

disturbance due to missing labels in the background	
Orientation in the shown map is easy.	37%
The missing labels at the back disturb me.	0%
The missing labels at the back do not disturb me but I would prefer a complete labeling.	58%
undecided	5%

Table 2.4: Results for the question if missing labels in the upper part of the view trouble the user.

60% of the typos. This either indicates that there was a learning effect or perspective labels are easier to read. To the contrary, the participants starting with labels with perspective distortion found 58% of the errors in their first search process but only 56% in the labeling with regularly-rendered labels. The participants of this group possibly had a learning effect compared to the results for Figure 2.4 (regular) of the other group. Also the fact that in the first search process (independent from the group) no one could find all the typos but in the second search process three participants could find all the errors indicates that there indeed was a learning effect. This weakens the result of finding 56% of the typos in the second search process. Another point that supports the usefulness of the style with perspective distortion is that only one person could not find any typo in the style with perspective labels (second search process) but three participants could not find any error in the style with regularly-rendered labels (two in the first, one in the second search process). All in all, the participants found 59% of the typos in Figure 2.3 (perspective) and only 43% of the typos in Figure 2.4 (regular). Further, regarding aesthetics, we obtained the striking result that 79% of the participants prefer the style of Figure 2.3 (perspective). One participant abstained.

Our designed figures have some weaknesses, though. In Figure 2.3 (perspective) names with characters with descenders (for example, *g*) have a slightly larger font size than in Figure 2.4 (regular). Nevertheless, even without this mistake, labels with perspective distortion appear to be slightly larger than regularly-rendered labels. The figures used by Maass et al. [MJD07b] show the same effect. (The reason is comparable with the fact that a diagonal in a square is longer than an edge.) Moreover, it is possibly adverse that we attached the faulty labels to streets of different heights. Another point is that, from a psychological point of view, some errors might be easier to find than others. Possibly, it is easier to find a missing *t* than a missing *c* or just the other way round. When designing the figures we tried, however, to avoid that the spelling of a word indicates if the label could be faulty. On that account, we also use street names that seem to be misspelled. For example, we use *Ludvigweg* whereas the spelling with *w* (instead of *v*) would be much more common.

Despite these weaknesses, we still conclude that the style in Figure 2.3 (perspective) is not only more useful but even more aesthetic. This certainly justifies us in using perspective labels in maps that permit a 3D view. In effect, this means that we can apply algorithms that place embedded labels in a 2D view also for labeling maps in a 3D view without any changes. With these findings we give a first hint to the issue stated by Maass et al. [MJD07b]; they state that, so far, there is only little known about the impact of the two rendering possibilities. Certainly, we have to verify our results in a user study with moving pictures.

As Table 2.4 shows, none of the participants was disturbed by the missing labels in the background; only one participant abstained. Nevertheless, 61% of those who had no problem with the partly labeled map, would prefer a completely labeled map. We believe, however, that our results would differ if we had designed and shown a figure with a complete labeling that, due to the 3D view, is hardly legible in the upper part of the view; in other words, we expect that the results would differ if we would have shown a figure for comparisons. We conclude that, in the case of hardly-legible labels in the upper part of the view, it is no problem to omit these labels.

2.4 Concluding Remarks

We have conducted a preliminary user study based on static pictures. We first considered the problem of attaching regularly-rendered labels to point features in a 3D view. We came to the conclusion that, starting in a 2D view, all labels should have the same height; when tilting, labels shrink with their distances to the user.

The second part of our study deals with the problem of labeling streets in maps with a 3D view. We posed the question if it is better to attach billboards or embedded labels to a route that leads the user to a target. The participants were undecided. On that account, we concluded that labeling streets with billboards is a concept worth trying.

When we conducted this study, we rather aimed for developing algorithms for navigational devices than for digital maps in general. From where we stand, the study should have been related to the more general case of digital maps. There also have been other weaknesses in our tests. Nevertheless, we do not doubt our results. With our study we wanted to know if our ideas are reasonable and, if so, in which order we should investigate them. As our preliminary study was based on steady pictures, it is anyhow indispensable to conduct a study using moving pictures in order to verify the aesthetics and the usefulness of our approaches. In the study using moving pictures, we could test how long the user needs to find a certain label. Possibly, we could also make some tests in a driving simulator. While users are searching for their ways in an unknown environment, suddenly a car passes from the left to the right. When we measure the reaction time, we get to know how much the labeling distracted the drivers or how intensive they had to search their ways.

It would also be interesting to extend the result of Bachfischer [Bac05]. How much overlap does users accept in a 3D view?

When we conducted the preliminary study, user studies were a quite new field for us. For the final study, it would be advisable to cooperate with psychologists. Moreover, we should first publish our ideas about the study before conducting it. Today, we know that this is a typical way and several of our mistakes could have been avoided that way.

Part I

Labeling Point Features in Interactive Maps

Chapter 3

Optimizing Active Ranges for Labeling Point Features

Although the problem of automatically labeling point features can be considered solved for static maps, there is not much work done for interactive maps. We investigate the problem of labeling point features with disk-shaped labels in a dynamic setting where the user can manipulate the view by panning, zooming, and rotation operations. Such disk-shapeds label could, for example, represent pictorial labels that are attached to POIs. Our labels are centered at their reference points. Consequently, it suffices to update the labeling while the user zooms. When the user zooms, labels maintain their sizes on the screen. Note that with such labels and when always computing the labeling for the entire map, it suffices to update (and thus to concentrate on) the labeling while zooming. We expect, however, that every appearance or disappearance of a label during a zooming operation might distract the user. In the worst case, labels flicker.

In this chapter, we propose algorithms for the following problem. Given a set of points of interest (for example, gas stations or parking lots), for each map scale, select a large subset of points that can be labeled such that no two labels overlap and no label flickers.

Our Model. In our dynamic setting, the user can continuously pan, zoom, and rotate a map in a 2D view. In order to model the behavior of labels while zooming, we consider *active ranges*, a concept that has earlier been introduced by Been et al. [BNPW10]. Realize that a map of scale $1 : z$ has scale factor z. Let $\ell(p)$ be the label of the reference point p. In our model, the active range of a label $\ell(p)$ is an interval $\mathcal{A}(p)$ of scale factors such that, for any scale factor $z \in \mathcal{A}(p)$, in a map at scale $1 : z$, the point p is labeled. (For a better readability, we use the notion $\mathcal{A}(p)$ instead of the more precise notion $\mathcal{A}(\ell(p))$.) We can formalize our point-labeling problem *active range optimization* (ARO) as follows.

Given a set P of points in the plane, the constant $d > 0$ which represents the diameter of a label at the screen, and the largest scale factor $z_{\max}$ at which the map is displayed, assign active ranges to the labels such that

(C1) the active range $\mathcal{A}(p)$ of each label $\ell(p), p \in P$, consists of exactly one interval $(0, z(p)]$ with $z(p) \leq z_{\max}$,

(C2) in a map of scale $1 : z$, no pair of labels $\ell(p)$ and $\ell(q)$, $p, q \in P$, whose active ranges contain z overlap, and

(C3) the overall length of all active ranges is as large as possible.

We require (C1) as a hard constraint to avoid that a label flickers. To ensure that all labels are placed when zooming in far enough (that is, when the scale approaches infinity), we require $\mathcal{A}(p) = (0, z(p)]$ for each label $\ell(p)$, $p \in P$. On the other hand, when we zoom out to a scale smaller than $1 : z(p)$, the label $\ell(p)$ disappears.

Obviously, Constraint (C2) is a *legibility* constraint. Moreover, our problem statement includes a *preservation* constraint, namely Constraint (C3): Constraint (C2) makes labels disappear whereas Constraint (C3), tries to keep points labeled. Apparently, if we zoom out to a quite small scale, we cannot keep all points labeled while satisfying Constraint (C2). Therefore, our problem statement defines the preservation constraint to be soft, that is, we maximize the overall lengths of the active ranges.

Note that the active ranges are a data structure that we compute for the entire map in a preprocessing, that is, we compute the labeling offline. At runtime, we simply query the labels to be placed. Consequently, our algorithms are real-time capable. We anticipate that our algorithms for building the active range data structure can easily be adapted in order to deal with other geometric shapes, for example, rectangles. We further remark that we can change the point of view: instead of zooming in and out, we could consider a map of constant scale and grow and shrink labels suitable. At the largest scale, that is, for $z = 0$, labels degenerate to points.

A Generalization Problem. On the one hand, maps for car navigation systems need to display sufficient information to allow their users to orientate themselves; on the other hand, since driving requires attention to the road, such maps must not distract the users with too many details. That is, the maps need to be generalized. (In our approaches, we usually require to place as many labels as possible. This is no contradiction, though. As stated in the introduction, we simply can omit some labels.)

Our version of ARO directly corresponds to a generalization problem, namely the problem of selection. Instead of displaying all reference points and placing disk-shaped labels, we could consider the problem of selecting a set of points to be displayed such that each pair of points has a minimum distance. To this end, we assign active ranges to points (instead of labels) and we require that, for each pair $\{p, q\} \subseteq P$ of displayed points, p and q have a distance of at least d (which directly corresponds to the constraint that two labels with diameter d must not overlap).

Selection is generally considered as one of a small number of map generalization operators; for example, one of the eight operators in the classification Beard and Mackaness [BM91]. For a long time, research on map generalization focused on static topographic maps. Our approach, however, is designed for generalizing interactive maps which allow for zooming, panning, and rotation operations. As for the labeling variant of this problem, zooming is the only operation that needs to be considered.

Constraint-based approaches are deemed as promising for generalization [WD98]. We remark that most constraint-based models for map generalization include a legibility constraint (Constraint (C2)) and a preservation constraint (Constraint (C3)) [BSS07]. Also the concept of building a data structure for the entire map in a preprocessing and querying data at runtime, is a typical approach. Van Oosterom [vO95], for example, aggregates area objects and stores the results in a data structure, the GAP-tree. At runtime, users can request maps of arbitrary scales.

We can combine the generalization and the labeling approach by, for example, setting $d^* = d + k$, where d is the diameter of a label and k an additional distance between two labels.

Our Contribution. We first show how to formulate ARO as a *(mixed) integer linear program (MIP)* (see Section 3.2). Integer programming is a global optimization technique. In general, solving a MIP is NP-hard but there exist highly optimized commercial and non-commercial MIP solvers. To readers who are not familiar with this optimization technique, we recommend an article of Haunert and Wolff [HW10]. They apply integer programming to the generalization problem of area aggregation.

Second, the optimal solutions that we obtain by solving the linear program allow us to assess the quality offered by more efficient heuristic methods. This helps us choose a good heuristic for large problem instances. In fact, to solve ARO efficiently (but not necessarily optimally) we present a new heuristic, the *growing-cones heuristic* (see Figure 3.1 and Section 3.3 for example outputs). We give two variants. Both variants of the heuristic exploit that the Delaunay triangulation of the given point set contains, at any time, an edge between a closest pair. Then, we compare the solutions of our heuristic to those of the MIP and a heuristic that have been introduced by Been et al. [BNPW10] (see Section 3.4). We run our tests on both real-world and synthetic data. We finally conclude this chapter by presenting some ideas concerning the weighted case, obstacles at the map, and accumulating labels (see Section 3.5).

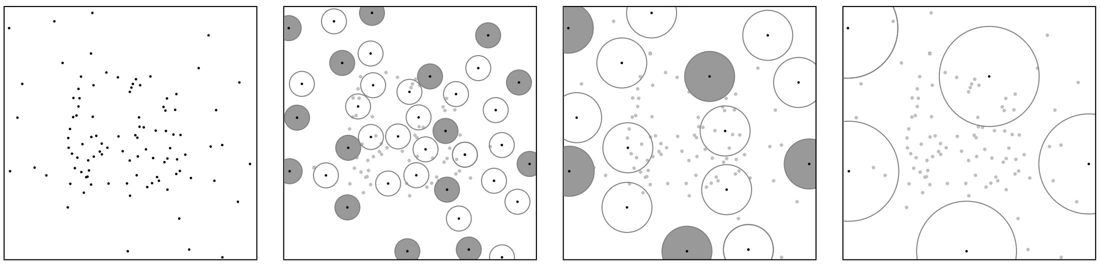

(a) $\delta = 0$; 100 points (b) $\delta = 0.1$; 37 points (c) $\delta = 0.2$; 15 points (d) $\delta = 0.4$; 5 points

Figure 3.1: One of our heuristics (M1) applied to a set of 100 reference points; one quarter of the points is drawn randomly from the unit square $[0,1]^2$, the rest is drawn from the square $[1/4, 3/4]^2$. The diameter of the labels is δ. We shaded the labels that survive to the next scale.

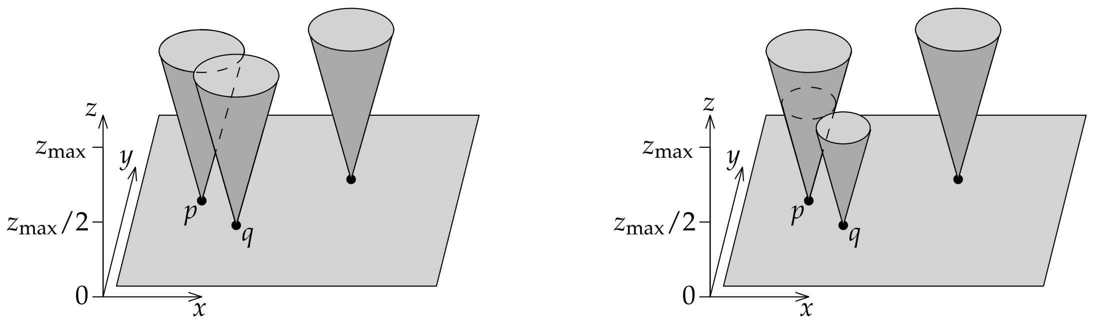
3.1 Previous and Related Work

As stated in the introduction, some digital maps are organized in LoDs. Poon and Shin [PS05] propose an approach for selecting the labels for one LoD after zooming.

They compute the labeling of the current scale based on the data of the next *larger* scale. They give an algorithm to suitably select the labels to be placed such that the number of placed labels is large and no two labels overlap. Their labeling is, however, not consistent.

Been et al. [BDY06] present a data structure for displaying consistently labeled maps under panning and zooming operations. They use rectangular labels. In order to avoid that labels jump they simply insist that the position of a label relative to its reference point remains the same over all scales. This permits them to use a data structure that represents each label by a pyramid whose apex coincides with the reference point at $z = 0$. The objective of Been et al. is to compute a set of pairwise (interior-) disjoint frusta, at most one per pyramid, whose total height is maximized.

In our model, we pursue the same idea: our disk-shaped labels are centered at their reference points, that is, on the one hand, label positions relative to their reference points remain the same over all scales; on the other hand, we can represent each label $\ell(p), p \in P$, by a cone $c(p)$ whose apex coincides with the reference point at $z = 0$; see Figure 3.2. Our objective slightly differs in that we require that each cone starts at $z = 0$.



(a) an infeasible solution; cones $c(p)$ and $c(q)$ intersect

(b) a feasible solution obtained by truncating $c(q)$

Figure 3.2: Interpreting the scale as the third dimension (in addition to the two spatial dimensions x and y), disk-shaped labels are cones. In an overlap-free labeling, no two cones intersect. Note that, even if two circles intersect in the top view, the corresponding cones do not necessarily intersect in a side view.

Using the data structure of Been et al. [BDY06], the labeling during or after a user interaction can simply be obtained by intersecting the precomputed set of frusta with the horizontal rectangle that corresponds to the part of the map that the user currently sees. Hence, their data structure allows for real-time interactions. Been et al. present a simple heuristic for consistent map labeling and show that the problem is NP-hard in the more general case where the frusta are not restricted to pyramids, but to frusta (which means that users can specify that labels are visible only within a certain scale interval).

Been et al. [BNPW10] refine this work from a more theoretical point of view. They show that the problem remains NP-hard in the original frusta-in-pyramids setting.

Their proof carries over to our version of ARO, that is, the case of disk-shaped labels. The authors also present two algorithms for ARO dealing with congruent square cones. The first algorithm has an approximation *factor* of 4, which means that the set of frusta it computes has a total height of at least $OPT/4$, where OPT is the total height of an optimal solution (which is NP-hard to compute). The algorithm runs in $\mathcal{O}((k+n)\log^3 n)$ time, where n is the number of reference points and k is the number of pairs of intersecting cones. From their proofs, we can deduce that their algorithm applied to disk-shaped labels, that is, to our version of ARO, has an approximation factor of 6. This is due to the fact that we can arrange at most six *open* disks of the same size around another open disk of the same size such that there is no pair of disks that intersects (see Figure 3.3); a disk is open if the circle that represents the boundary of the disk is not considered part of the disk. If we require that labels must not intersect at their boundaries, that is, if we require *closed* disks, we even obtain an approximation factor of 5. The second algorithm that Been et al. introduce has an approximation factor of $(4+\varepsilon)$ for any $\varepsilon > 0$ and runs in $\mathcal{O}(n \log n \cdot \log(n/\varepsilon)/\varepsilon)$ time. The second algorithm is faster than the first if the number of pairs of intersecting cones is large (say $k = \Theta(n^2)$) and ε is large (say $\varepsilon = 1$).

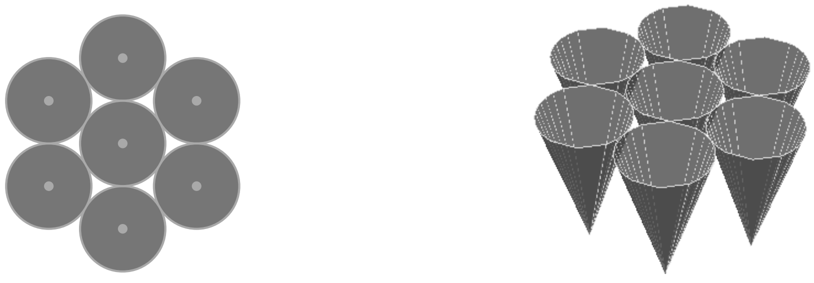

(a) top view (b) bird view

Figure 3.3: A disk touches at most six disjoint disks of equal size.

Unlike disk-shaped labels, rectangular labels can get overlapped if the user rotates the view. While Been et al. [BDY06, BNPW10] focus on panning and zooming interactions, Gemsa et al. [GNR11a] consider rotation, thus complementing the work of Been et al. Similar to Been et al., they compute, for each given point, an active range (now an interval in $[0°, 360°)$) for the corresponding label such that, when rotating the view by an angle α, the (axis-parallel congruent square) labels of all points whose range contains α are pairwise disjoint. The authors show that, on the one hand, this version of ARO is NP-hard; on the other hand, near-optimal solution can be computed efficiently. Gemsa et al. [GNR14] verify these theoretical findings by means of practical experiments.

Gemsa et al. [GNR11b] introduce a 1D-version of ARO where the input points all lie on the same horizontal line. The lower edge of a label can slide along its point. Labels must not overlap. The aim is to select a single position for every label (which is then fixed over all scales) so that the sum of the lengths of the active ranges is maximized. The authors solve the problem optimally for the case that every label can be placed at a given number q of discrete positions. Their algorithm runs in in $\mathcal{O}(n^3q^3)$ time. For the case that labels can touch their points arbitrarily, they present a *polynomial-time approximation scheme* (PTAS), that is, for any $\varepsilon > 0$, their algorithm computes an approximation with factor $(1 + \varepsilon)$ whereas the algorithm needs $\mathcal{O}(n^3/\varepsilon^3)$ time.

Gemsa et al. [GNN13] also introduce active ranges of time. They assume that the user follows a route which is completely known in advance. While passing the route, the labeling should be overlap free and, in total, maximize the sum over the time intervals of all labels.

Nöllenburg et al. [NPS10] investigate panning and zooming operations, but for boundary (or rather margin) labeling, where labels are placed in the margin of the map and then connected to the given points by leaders that are polygonal lines. The authors use rectilinear leaders with at most one bend. Given an interval of scales, they compute, for each scale, label and leader positions such that the total leader length is minimized. Their algorithm is efficient and allows for real-time interactions.

The existing approaches based on active ranges allow one degree of freedom, that is, scale, rotation angle, *or* time. It may be possible to deal with active ranges of higher dimensions, but, since current digital maps allow for zooming, rotation, panning, *and* tilting operations, we doubt that the problem of labeling interactive maps can be solved with the help of precomputed active ranges alone. On that account, we restrict our approach to scale, too.

Kovanen and Sarjakoski [KS13] argue that using a data structure holding preprocessed data for the entire map is not always reasonable. For example, sometimes, it is not possible for the user to download the data to be visualized from the Internet. To this end, they present an efficient algorithm that attaches square labels to points of interest while avoiding label–label overlaps in *one* scale. Their algorithm does not depend on the labeling *history*, that is, the labeling of the current frame does no take the labeling of the preceding frame into account but their algorithm computes a new labeling from scratch. The algorithm processes incrementally: for each label ℓ to be placed, it first tests if ℓ overlaps a label that is already placed. If so, the algorithm tests if a label with the same symbol is already included in the group of labels that the overlapped label belongs to. If not, the algorithm computes label candidates around the group. If there is an overlap-free position for ℓ, ℓ is included in the group. Otherwise, the algorithm indicates the missing label by placing the number of omitted labels at the lower right corner of a label of the group. The authors implemented their approach. They state that, on an iPad 2 tablet, their algorithm needs less than 200 milliseconds for placing 200 labels. The algorithm considers weighted reference points but it does not take into account the history of the labeling or obstacles at the map background.

Gerrits [Ger13] deals with the problem of labeling dynamic point sets. On the one hand, he shows that all natural optimization problems for dynamically labeling point features are PSPACE-hard, this is, they are at least as hard as NP-hard problems. On the other hand, he formulates the new objective of *free-label maximization* where the goal is to maximize the number of non-overlapping labels. For the static case and unit-square labels, he presents an $\mathcal{O}(n \log n)$-time algorithm whose approximation factor is between 7 and 32, depending on the number of allowed label positions relative to the reference point. He uses this approach as a subroutine for labeling dynamic point sets, this is, sets in which points can be added and removed, and points can move on continuous trajectories. Instead of proving an approximation factor, he processes experiments to show that his algorithm works well in practice.

Das Sarma et al. [SLG^{+}13] present three algorithms for selecting points to be displayed. Their objective is similar to ours: their require for Constraint (C1) and (C3) but not for Constraint (C2). The authors store the points in a quadtree where the root represents the smallest scale, that is, the entire map, and the leaves represent the map in the largest scale. Consequently, the authors maintain LoDs. Das Sarma et al. require that the displayed points are thinned out at each node of the quadtree such that the number of displayed points is bounded by a constant K. They give a linear program solving the problem optimally. (Linear programs equal integer linear programs whereas the variables can take on any irrational number; they are efficiently solvable). Further, the authors give a heuristic that is similar to the heuristic introduced by Been et al. [BNPW10] (we introduce a variant of the approach of Been et al. in Section 3.3.1). Instead of using a breadth-first search according the heights of the pyramids (as Been et al.), Das Sarma et al. apply a depth-first search to points in the quadtree. While traversing the tree, they select points such that K is satisfied. The results of both these approaches, that is, the linear program and the depth-first search, have to be stored and queried. Das Sarma et al. give another algorithm that selects the points to be displayed on the fly. The authors assign a random but unique number to each point. Then, for an arbitrary node in the quadtree, they just display the K points with the highest numbers. The authors also implemented their algorithms. They conclude that the linear program processes 10,000 points in about 5 minutes; their approach based on the depth-first search processes point sets of up to 20 million points in about 8 minutes; the approach using random numbers is able to handle arbitrarily large data sets. The difference of the values of the objective function of both heuristics compared to an optimal solution is only about 1%.

3.2 A Mixed-Integer Linear Program

In general, mixed-integer programming allows us to solve small problem instances with proof of optimality. This implies that the cartographic quality of the output only depends on whether or not we have modeled all relevant aspects of the problem appropriately. In particular, the output is *not* influenced by tuning parameters, which are common to most heuristic optimization methods. This has two advantages. First, we can verify the appropriateness of our model based on optimal solutions that we obtain for small instances. Second, such solutions allow us to assess the quality offered by more efficient heuristic methods.

The NP-hardness of Aro [BNPW10] justifies the application of a mixed-integer linear program (MIP). Linear programming is a technique for describing an optimization problem using linear terms only. Any linear program consists of (i) variables, (ii) an object function, and (iii) constraints. In a MIP, some variables are continuous, some are integer-valued.

Our MIP formulation is as follows. We introduce, for each $\ell(p), p \in P$, a *continuous variable*

$$z_p \in (0, z_{\max}]$$

that encodes the upper bound of the active range of $\ell(p)$. Now, according to Constraint (C3), our *objective* is to

$$\text{maximize} \sum_{\ell(p), p \in P} z_p.$$

Further, for each pair $\ell(p), \ell(q),\ p, q \in P$, of labels, we introduce the *binary variable*

$$y_{pq} \in \{0, 1\}$$

which is 1 if $z_q \leq z_p$ and 0 otherwise. The interpretation of $y_{pq} = 1$ is that, when zooming out, $\ell(p)$ is longer visible than $\ell(q)$. We observe, if $z_q \leq z_p$, the distance $d_{z_q}(p,q)$ of p and q at the screen in a map of scale $1 : z_q$ is at least d; both labels are placed. Clearly, $d_{z_q}(p,q) = d_{\text{map}}(p,q)/z_q$, where $d_{\text{map}}(p,q)$ is the distance of p and q at the map. This yields $z_q \leq d_{\text{map}}(p,q)/d$. The implication "$z_q \leq z_p \ \Rightarrow\ z_q \leq d_{\text{map}}(p,q)/d$" can be expressed by the following two *constraints*, which we introduce for each pair $\ell(p), \ell(q),\ p, q \in P$, of labels:

$$z_p \leq \frac{d_{\text{map}}(p,q)}{d} + z_{\max} \cdot y_{pq} \quad \text{and}$$

$$z_q \leq \frac{d_{\text{map}}(p,q)}{d} + z_{\max} \cdot (1 - y_{pq}).$$

Note that the first constraint does not have any effect if $y_{pq} = 1$ (since it holds anyway that $z_p \leq z_{\max}$). Similarly, the second constraint does not have any effect if $y_{pq} = 0$. In words, the MIP restricts for each pair $\ell(p), \ell(q)$ of labels, either z_p or z_q, that is, the scale factor at which $\ell(p)$ or $\ell(q)$ is removed when zooming out, such that the distance on the screen between the corresponding reference points p and q is at least as large as the diameter d of a label. Consequently, placed labels are overlap-free. Simultaneously, we require that each scale factor is as large as possible (otherwise also $z_p = 0$ for each $\ell(p),\ p \in P$, is a feasible solution). We conclude the presentation of our MIP with the observation that the constraints are linear since $d_{\text{map}}(p,q)$, d, and $z_{\max}$ are constants (that is, not variables) from the point of view of the MIP.

3.3 Greedy Algorithms

Due to the unpredictable computation time of a MIP solver, we present two simple heuristics that are based on a greedy strategy, that is, once the algorithm made a decision, it cannot be reverted. Both heuristics are efficient; the results of the first are at most by a factor of 6 worse than the optimum (which is NP-hard to compute), see the argument in Section 3.1. We can use these heuristics for computations that rather aim for a fast computation than an exact computation, for example, for processing large data sets or for applications with strong requirements in terms of computation time.

3.3.1 Shrinking-Cones Heuristic

Algorithm 3.1 was introduced by Been et al. [BNPW10]. Originally, the authors applied it to square labels (in scale space: square pyramids instead of cones). Recall that $z(p)$ is the upper bound of the active range $\mathcal{A}(p)$ of the label $\ell(p)$. Let $c(p)$ be the cone in scale space with apex p whose base is a horizontal circle of diameter $d \cdot z(p)$ around the projection of p to the plane $z = z(p)$; see Figure 3.2(a).

Algorithm 3.1: ShrinkingCones(P)

```
Input: point set $P$
Output: for each label $\ell(p),\ p \in P$, the upper bound $z(p)$ of its active range
let $\mathcal{D}$ be a dynamic data structure that stores cones according to their heights
foreach $p \in P$ do
    $z(p) \leftarrow z_{\max}$
    $\mathcal{D}$.insert($c(p)$)
while $\mathcal{D} \neq \emptyset$ do
    $c(p) \leftarrow \mathcal{D}$.extractMax()
    foreach cone $c(q) \neq c(p)$ that overlaps $c(p)$ do
        truncate $c(q)$ such that $c(p) \cap c(q) = \emptyset$
            // that is, set $z(q) \leftarrow d_{\mathtt{map}}(p,q)/d$; see Fig. 3.2(b)
        $\mathcal{D}$.decreaseKey($c(q)$)
```

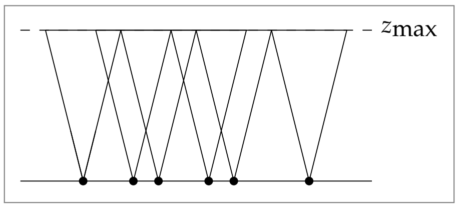
The idea of this algorithm is to start with cones of height $z_{\max}$. These, of course, may overlap. Therefore, step by step, we choose a cone $c(p)$ of maximum height that we have not yet fixed and shrink all the cones that overlap $c(p)$. Now, we consider $c(p)$ as fixed and part of our solution. If there are at least two cones of the same height, we choose the cone to be truncated arbitrarily.

Figure 3.4 illustrates an example for the shrinking-cones heuristic of Algorithm 3.1. Note that the algorithm does not necessarily compute an optimal solution (see Figure 3.4(j)) but, when fixing the cones in a suitable order, the heuristic could have computed it. Further, Figure 3.5 visualizes the output of Algorithm 3.1 applied to the real world instance shown in Figure 3.6.

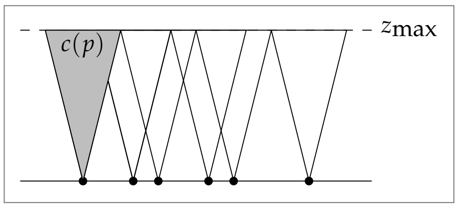
When using a conflict graph to maintain pairs of intersecting cones and a heap that stores cones according to their height (data structure $\mathcal{D}$ in Algorithm 3.1), the algorithm runs in $\mathcal{O}((k+n)\log n)$ time, where $n = |P|$ and k is the number of pairs of intersecting cones (of height $z_{\max}$). This results from an adjustment of the Bentley–Ottmann sweep line algorithm for line segments [dBCvKO08] to the case of circular arcs.

3.3.2 Growing-Cones Heuristic

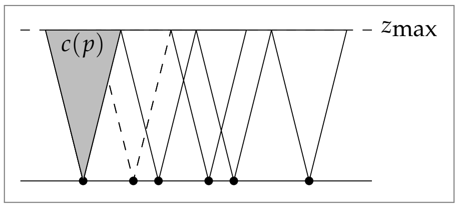
The idea of the second algorithm is to grow cones. We start with $z = 0$, that is, initially, each label is placed. Then, we repeatedly search for the smallest scale factor $z' > z$ at



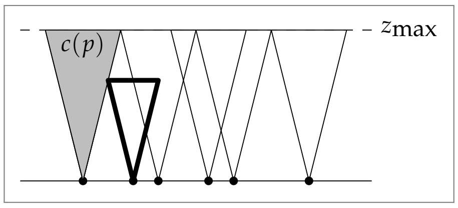
(a) start with cones of maximum height $z_{\max}$



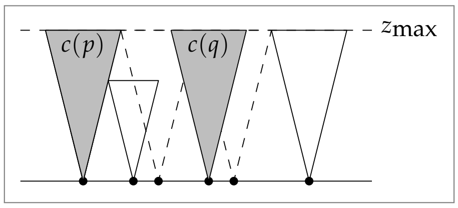
(b) fix the arbitrarily-chosen cone $c(p)$ (gray)



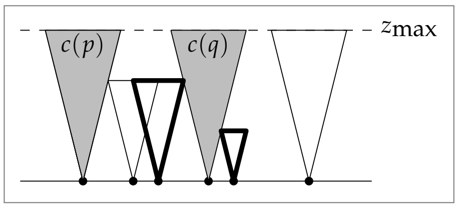
(c) determine cones that overlap $c(p)$ (dashed)



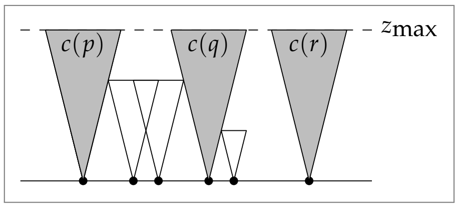
(d) truncate cones that overlap $c(p)$ (thicker outline)



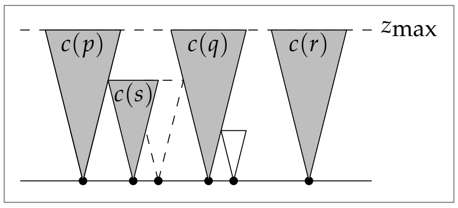
(e) according to the cones' heights, process $c(q)$



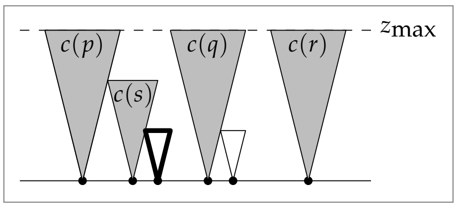
(f) truncate cones that overlap $c(q)$



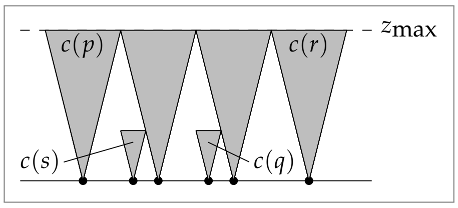
(g) fix the free cone $c(r)$



(h) fix cone $c(s)$

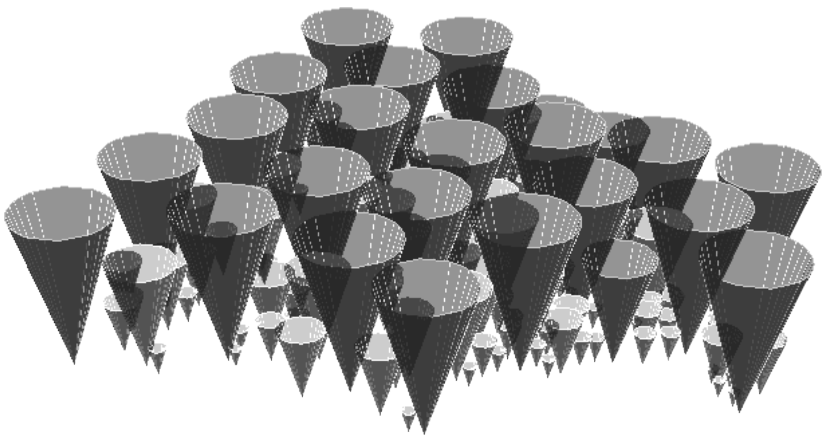


(i) truncate cones that overlap $c(s)$; fix the remaining cones

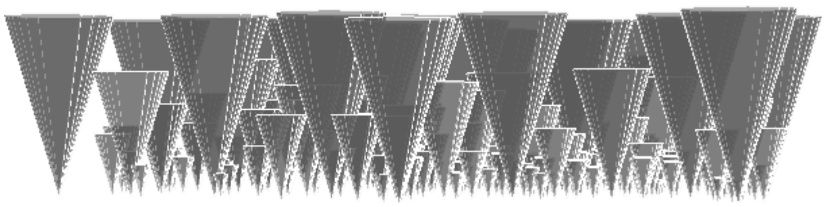


(j) optimum solution

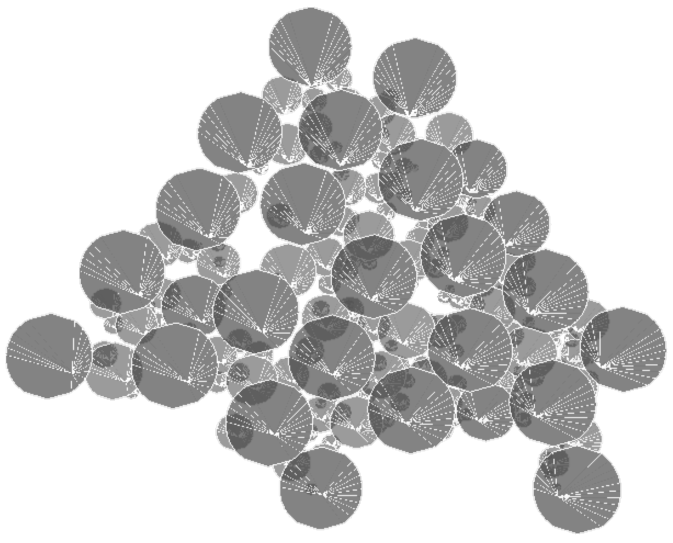
Figure 3.4: Illustration of our shrinking-cones heuristic of Algorihm 3.1. For the sake of simplicity, we used a one-dimensional point set where all points lie on a horizontal line and where we use triangles instead of cones for storing the active ranges.



(a) bird view



(b) side view



(c) top view

Figure 3.5: Perspective views of the result of the shrinking-cones method applied to the real-world instance in Figure 3.6. We added a transparency of 20% for cones whose active range is at least 70% of the maximum range.

which two labels touch. We remove one of the two touching labels, say $\ell(p)$, by setting $z(p) = z'$. We set $z = z'$ and continue until z reaches $z_{\max}$; see Figure 3.7.

We have implemented two methods for determining the label to be removed. Let p and q be the reference points of the touching labels; let $\ell(p)$ be the label of p. The first method, M0, arbitrarily removes one of the touching labels. The second method, M1, considers the reference points that are closest to p and q. It removes $\ell(p)$ if the distance of p and its closest neighbor (other than q) is smaller than the distance of q and its closest neighbor (other than p). With this, we remove the label that presumably will touch next.

In Figure 3.7, we use M1 (whereas, of course, the result could also be an output when applying M0). We observe that also M1 does not necessarily yield an optimal solution; see Figure 3.4(j). Nevertheless, it could. In the step shown in Figure 3.7(d), we first considered the touching cones on the left. As the distances of the corresponding reference points to their closest neighbors equal, we accidently stop growing the right-hand cone. Consequently, also the distances between the reference point of the touching cones on the right-hand side and their closest neighbors equal; thus, we choose the label to be removed, that is, the cone to fix, randomly.

Note that the closest pair of a point set forms an edge in the Delaunay triangulation. Therefore, we use the Delaunay triangulation to maintain the closest pair. Since the Delaunay triangulation has a linear number of edges, this immediately yields an implementation that runs in $\mathcal{O}(n^2)$ time for a set of n points. If we make the reasonable assumption that—in the process of deleting points—the average degree of a point involved in a closest pair in the Delaunay triangulation is constant, the running time of the algorithm reduces to $\mathcal{O}(n \log n)$ if we use a heap or a balanced binary tree to maintain the lengths of the edges in the Delaunay triangulation.

3.4 Experiments

In this section, we compare the results of our algorithms to those of the MIP. We implemented our heuristics of Section 3.3 and the MIP of Section 3.2 in C++. For the data structure $\mathcal{D}$ of Algorithm 3.1, we used a hash map. We assumed that labels are open disks, that is, labels are permitted to overlap at their boundaries. We executed our experiments on a Windows 7 system with a 2.3-GHz Intel quad-core processor and 4 GB of RAM. We applied the Microsoft Visual Studio 2010 Professional compiler in 32-bit release mode. For solving the linear program, we used Gurobi 5.1[3.1]. For the Delaunay triangulation, we used CGAL 4.1[3.2].

For our experiments, we considered two types of data sets. The first type is synthetic. For this, we picked points uniformly distributed in the unit square $[0,1]^2$. The second type is a real-world instance that consists of 249 points representing weather stations in the central part of the United States. We handled three series of tests. For the first series, *Series I*, we randomly generated point sets of size $n = 25, 50, \dots, 225$, and 249 in the unit

[3.1] `http://www.gurobi.com/`, accessed Mar. 30, 2013
[3.2] `http://www.cgal.org/`, accessed Oct. 18, 2012

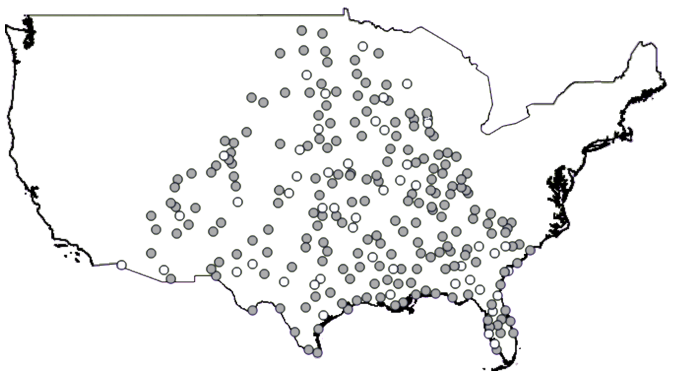
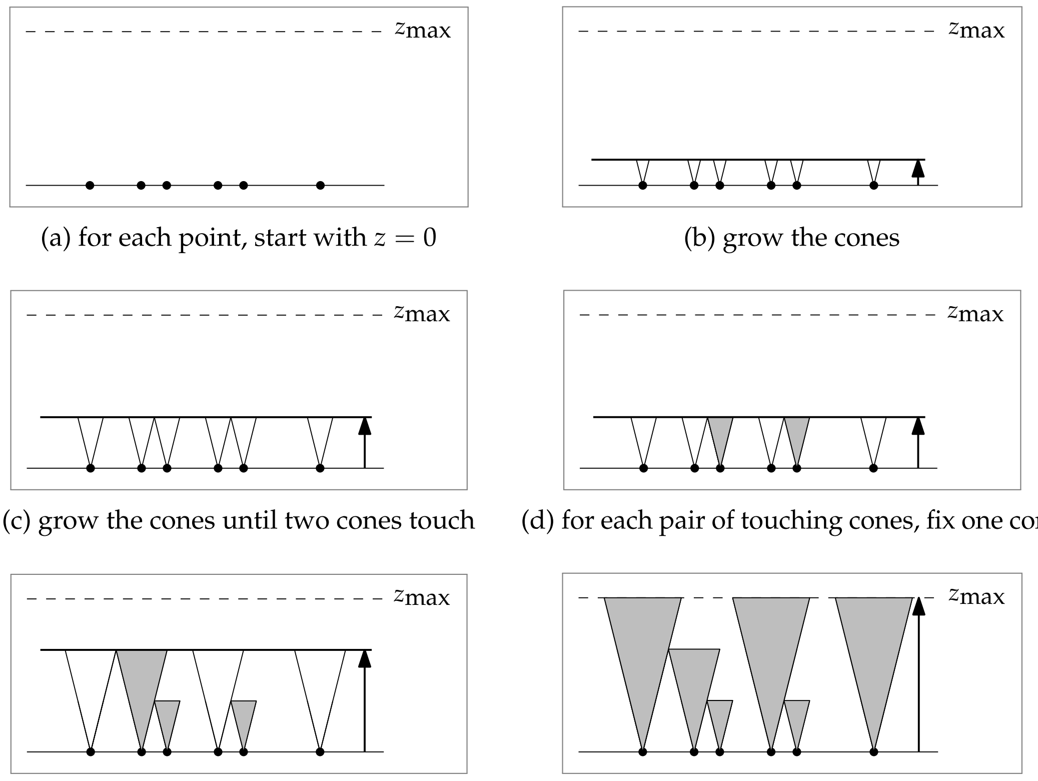


Figure 3.6: Our real-world instance consists of 249 weather stations in the central part of the United States. The gray points are 200 randomly-chosen points on which we applied all three heuristics and MIP35. We show the output in Figure 3.8.



(a) for each point, start with $z = 0$

(b) grow the cones

(c) grow the cones until two cones touch

(d) for each pair of touching cones, fix one cone

(e) grow further; fix a cone

(f) when a cone reaches $z_{\max}$, we are done

Figure 3.7: Illustration of our growing-cones heuristic. For the sake of simplicity, we used a one-dimensional point set where all points lie on a horizontal line and where we use triangles instead of cones for storing the active ranges.

square, five sets each. (We used 249 rather than 250 points in order that the synthetic data set matches the real-world instance.) For *Series II*, we used synthetic data, too. We generated point sets of size $n = 1000, 2000, \ldots, 5000$ points, three sets each. For Series I and Series II, on the map, the disks had a diameter up to $\delta = 0.2$. For *Series III*, we randomly generated point sets of size $n = 25, 50, \ldots, 225$, out of the real-world data set. We computed five sets each. Additionally, we used the set containing all $n = 249$ points. On the map, the disks had a diameter up to $\delta = 600$ kilometers (or about 375 miles). For each single set, we ran all our algorithms and recorded the computation time and the *performance*, that is, the value of our objective function. When running the MIP solver, we did not wait for optimum solutions but stopped the computation as soon as the solver could prove that the current solution is at most a pre-defined percentage worse than the optimum. We used a gap of 35% since after three days of computation the gap for a single set of 225 real-world points still was 36% (so, we even went without results for $n = 225$ and 249 for Series II and without any optimum result for Series III). We denote the MIP solver with a gap of 35% by *MIP35*.

Nevertheless, it is quite unusual that an optimum solution compared to the solution of MIP35 on the same data set differs by 35%. In order to verify this, we tried to compute optimum solutions at least for small point sets. For $n = 25$, the optimum result compared to MIP35 increases by 13%; the running time increases from 0.03 seconds to 0.51 seconds (this corresponds to a factor of 17). For $n = 50$, the optimum result increases by 16%; the running time increases from 0.11 seconds to 20 seconds (this is a factor of about 180). For $n = 75$, we tested only two point sets. Both runs were cancelled by the program because the program ran out of memory after 95 minutes of computations (compared to the computation time of MIP35 for 75 points, this is a factor of 20,200). The program stopped at a gap of about 3%; the result increased by 16%. These results confirm the choice of a gap of 35%.

Figure 3.8 shows the outputs of our algorithms for Series III at different scales. Figure 3.9 shows the performance of Series I and Series III relative to MIP35 and absolute values for the performance of Series II, averaged over the trials. Further it shows the absolute computation time averaged over the trials for all series of tests (one trial for $n = 249$ in Series III). Moreover, at the end of this chapter, Figures 3.11 and 3.12 depict some more point sets that were labeled by our algorithms. In these examples, the shrinking-cones heuristic outperforms the growing-cones heuristic, variant M1, in both running time and performance. Figure 3.9(c) indicates, however, that this was by chance.

The experiments show that the very simple shrinking-cones heuristic does astonishingly well. In terms of performance, even for the large point sets of Series I, the results of the shrinking-cones heuristic are at least 95% with respect to MIP35 in both data sets. Although we use a simple brute-force implementation (running in quadratic time), it is by a factor of 100–10,000 faster than the MIP solver with its gap.

For Series I and Series III, the growing-cones heuristic M1 is yet a few percent better than the shrinking-cones heuristic, but also by a factor of up to 10 slower. The shrinking-cones heuristic outperforms the growing-cones heuristic M0 in both performance and

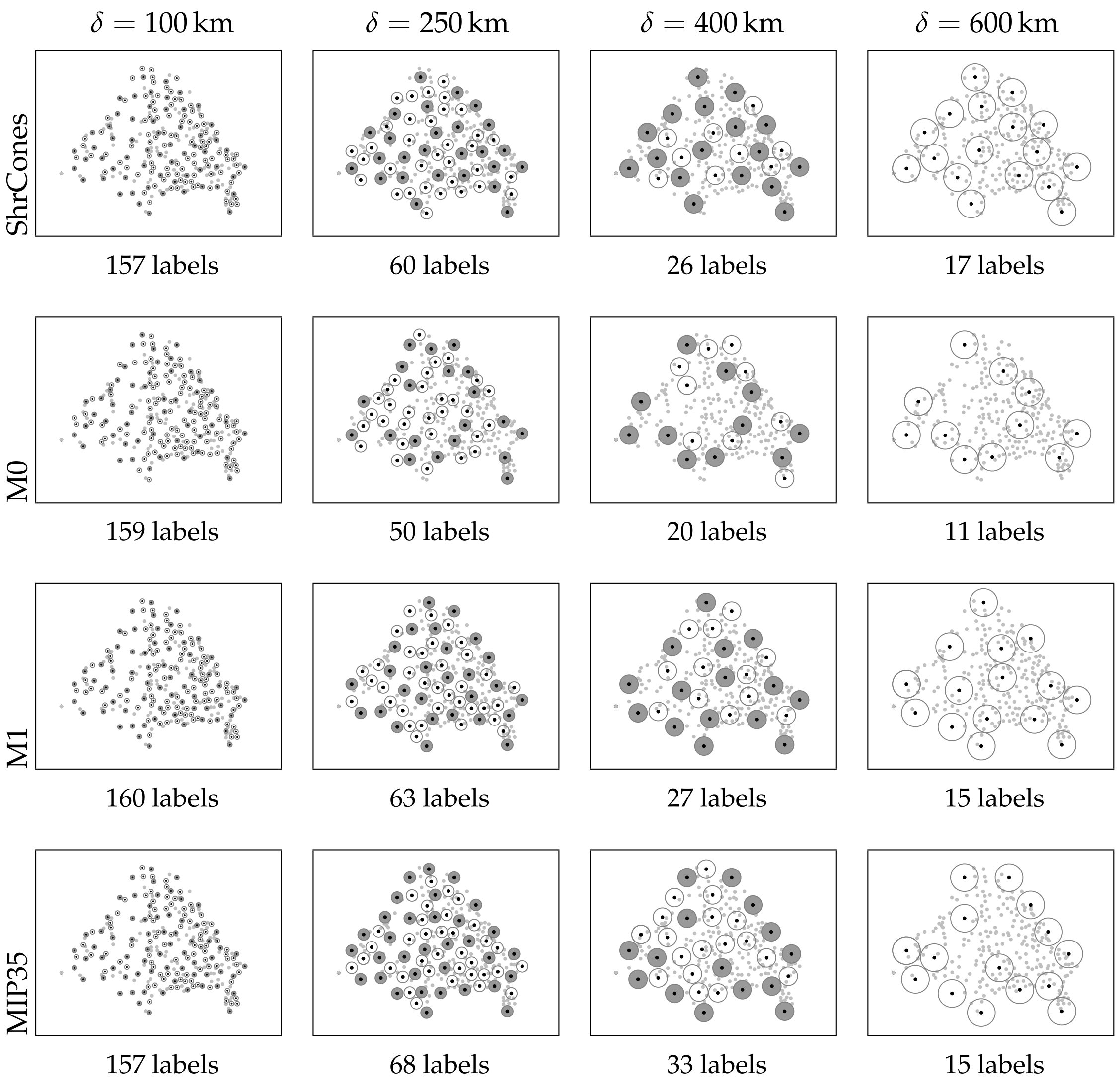
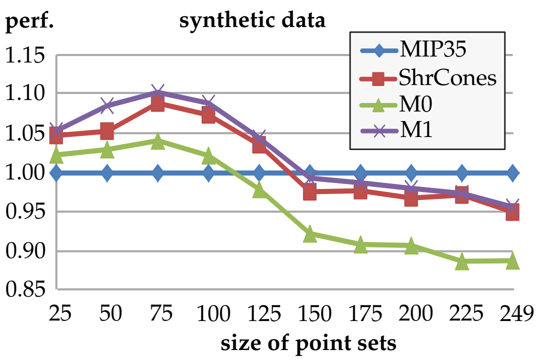
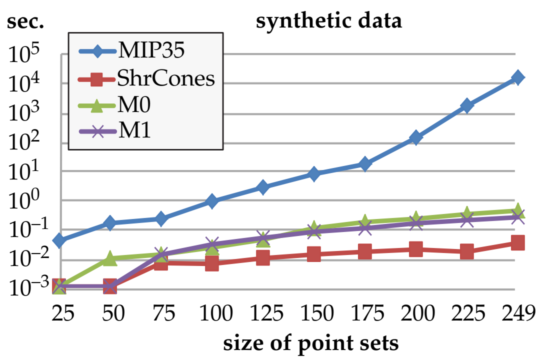


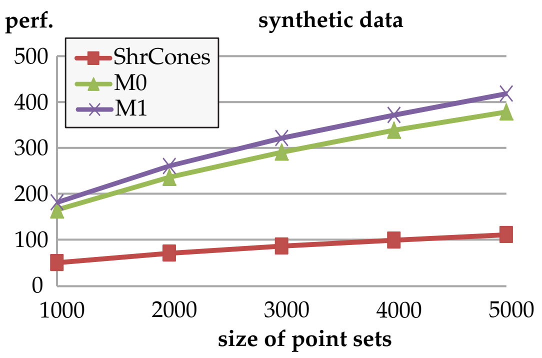
Figure 3.8: We applied all three heuristics and MIP35 to 200 points of our real-world instance of Figure 3.6. We show the results at different scales (grid). The diameter δ of the labels is given in the top row of the grid. We shaded the circles that survive to the next scale.



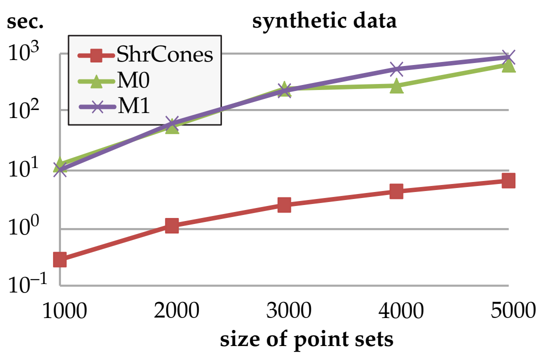
(a) total active range height relative to MIP35 for the synthetic data Series I



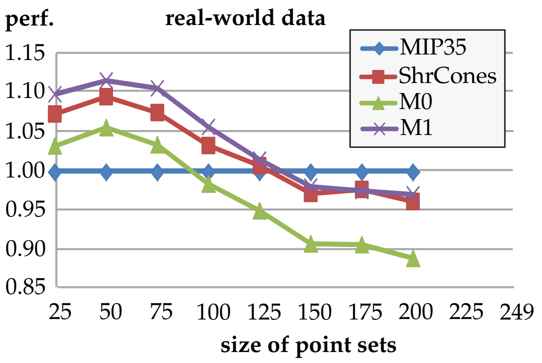
(b) absolute computation times (in seconds; log-scale!) for the synthetic data Series I



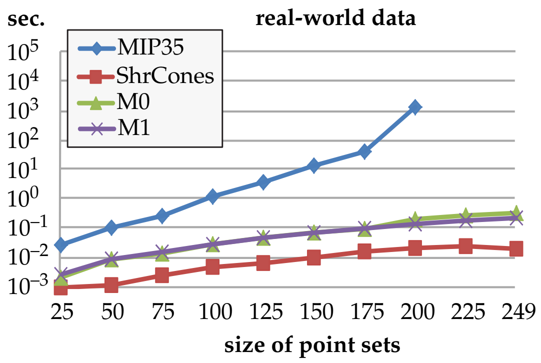
(c) absolute values of the total active range height for the synthetic data Series II



(d) absolute computation times (in seconds; log-scale!) for the synthetic data Series II



(e) total active range height relative to MIP35 for the real-world data Series III



(f) absolute computation times (in seconds; log-scale!) for the real-world data Series III

Figure 3.9: Performance and computation time for real-world and synthetic data set.

computation time. In Series II, on the one hand, both variants of the growing-cones heuristic outperform the shrinking-cones heuristic in terms of performance; on the other hand, the shrinking-cones heuristic is by a factor of 35–135 faster.

We conclude that, if performance is important, M1 is certainly the method of choice among the heuristics that we investigated. For the point set of 249 points, M1 takes 0.23 seconds on both settings with up to 249 point (Series I and Series III); for the synthetic point set of 5000 points it needs about 17 minutes.

3.5 Extensions

At last, we detail how our algorithms can be adapted such that they can deal with weights. Further, we present a concept of how to respect obstacles at the map. We finally give some ideas about the accumulation of labels.

Weighted Case. The data structure that we use in the shrinking-cones algorithm (Algorithm 3.1) sorts the cones according to height. If we want to take weights (that is, priorities) into account, we first sort by weight and then by height. This way, we first fix more important labels. We cannot truncate fixed elements any further. Therefore, the active ranges of important labels will tend to be larger than those of unimportant ones. This does not affect the asymptotic running time as the only difference to the original algorithm is the characteristic by which we sort the cones. The sum over all active ranges will change, though.

We can also adapt the algorithm that grows cones to deal with the weighted case. Whenever two labels touch, we have to decide which cone keeps growing and which one stops growing. We simply let the more important cone grow further. Obviously, this approach equals M0. Therefore, it has the same asymptotic running time. We expect that the sum of active ranges is smaller than that of M1 without weights.

Obstacles. The adaptation for dealing with obstacles is the same for all approaches. We can simply add suitable polyhedra (for example, pyramids) and treat them like cones that we cannot truncate, that is, they are fixed from the beginning. This will not affect the asymptotic running time.

Accumulating Labels. At the beginning of the thesis, we stated that we can preserve information by accumulating labels. Consider a zooming-out operation. Whenever a label ℓ vanishes, we accumulate it with the label ℓ^* (or the accumulation of labels) that made ℓ disappear; see Figures 3.10(a) and (b). We can visualize the accumulation, for example, by drawing a stack or placing the number of accumulated labels within the label symbol. At the top of the accumulation, we show either ℓ^* or the most important label of the accumulation. As only one label of the accumulation is visible, the user should have the option to spread the contained labels (see Figure 3.10(c)): when clicking at the top label of the accumulation, the contained labels should rearrange such

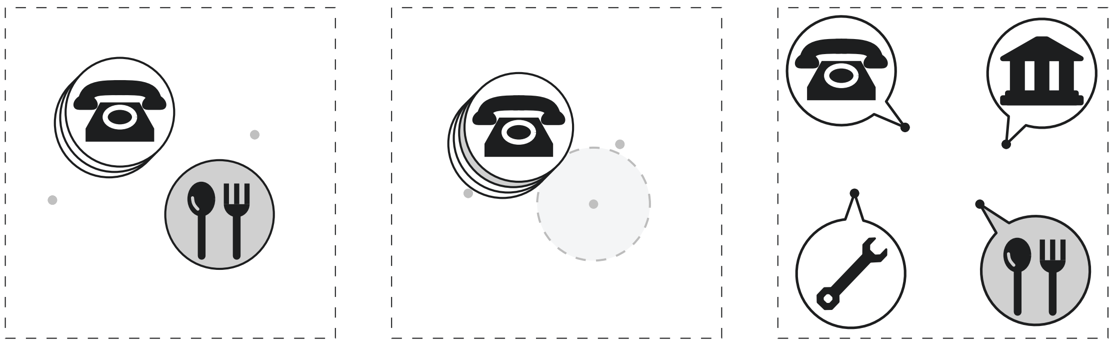

(a) situation before zooming out (b) we add the vanished label to the accumulation (c) labels of the accumulation spread

Figure 3.10: Accumulating and spreading labels.

that they are near to their corresponding reference points. Temporarily, labels of the accumulation should not overlap each other but they are permitted to overlap other labels or map objects. We maintain the correct label–object association for each label of the accumulation by connecting the label and its object, for instance, by a line. We should animate the rearrangements for generating smooth moving labels and prevent jumping labels. Similar ideas have been introduced by Fink et al. [FHS$^+$12] and Haunert and Hermes [HH14].

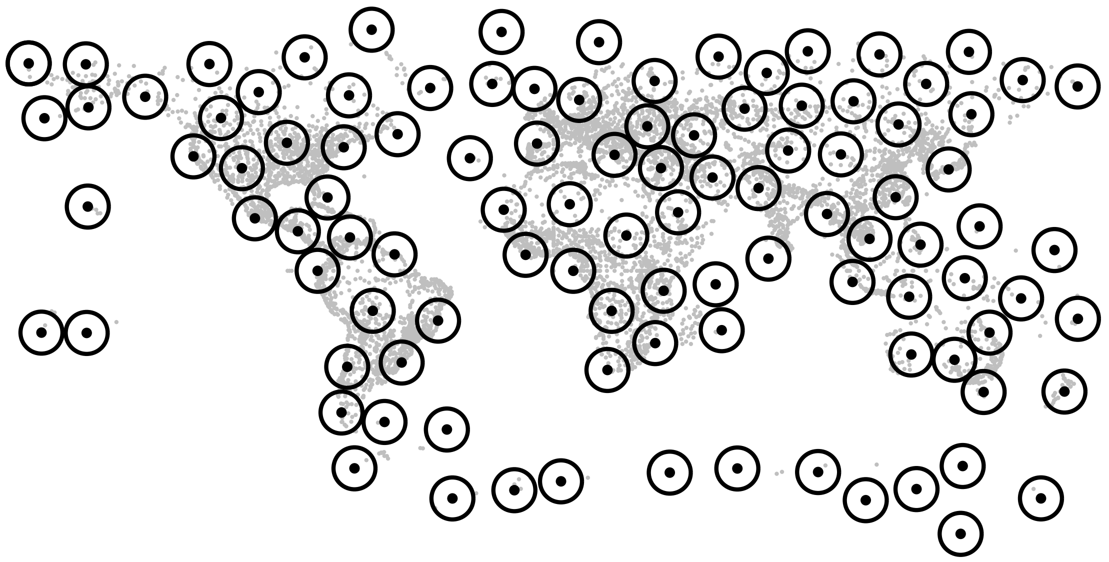
3.6 Concluding Remarks

We introduced different offline algorithms for Aro in interactive maps. Our MIP solves the problem optimally, but it is too slow for larger point sets. We also introduced efficient heuristics.

Concerning future work, it would be interesting to improve our implementation. Possibly, the shrinking-cones heuristic does better when we first sort labels by their distance according to the Delaunay triangulation and then by the amount of overlap. Moreover, the implementation should be able to handle the weighted case. It also would be nice to incorporate our approach into a navigation software.

Further, we can improve the value of our objective function if we permit to include frusta of cones, that is, cones might start at $z \neq 0$. That means, that at the largest scale of the map, possibly not all labels are visible. If we zoom out, labels might appear. Nevertheless, we still require that each label has only *one* active range. We think, however, that this is rather a theoretical approach in order to improve the value of the objective function than an approach that is relevant for practical purposes.

Finally, modern digital maps allow for perspective views. Although our approaches seem to work great in maps with 2D views, they have to be refined for labeling maps with a 3D view. We conclude that the matter of tilting requires more complicated shapes than just cones.


(a) shrinking-cones heuristic: 104 points, 19 seconds

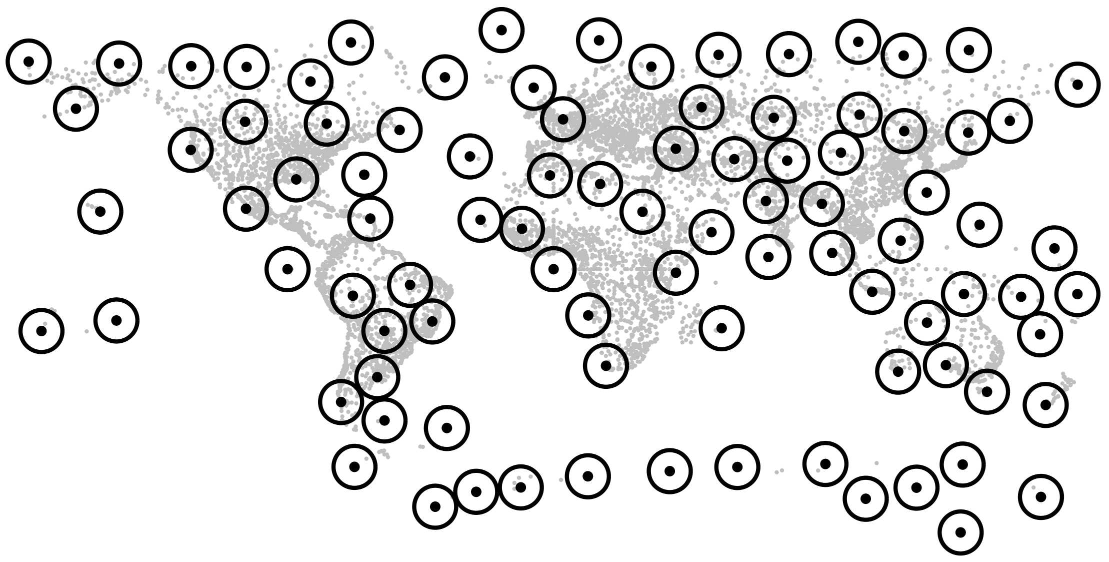
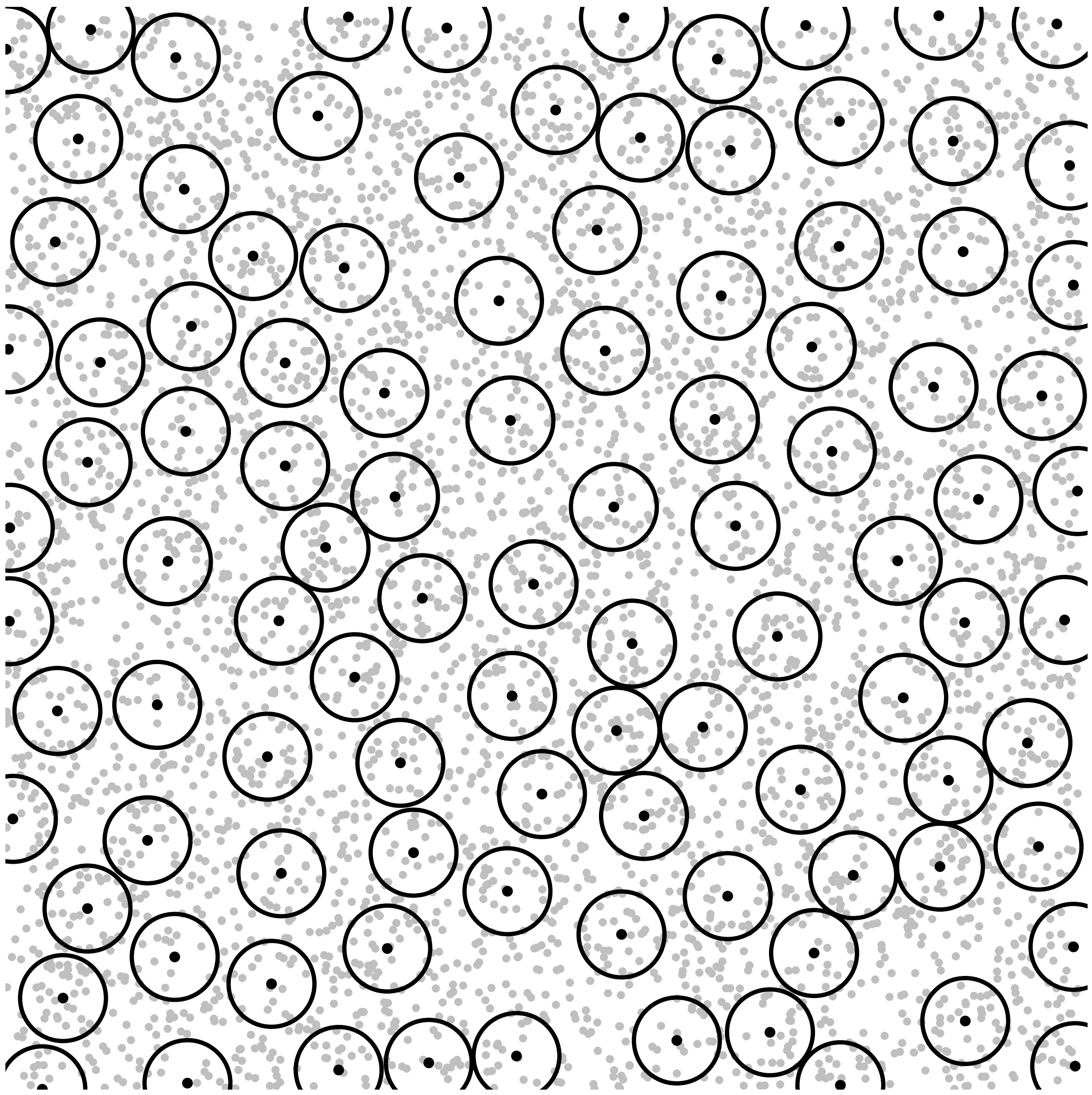

(b) growing-cones heuristic, variant M1: 92 points, 26 minutes

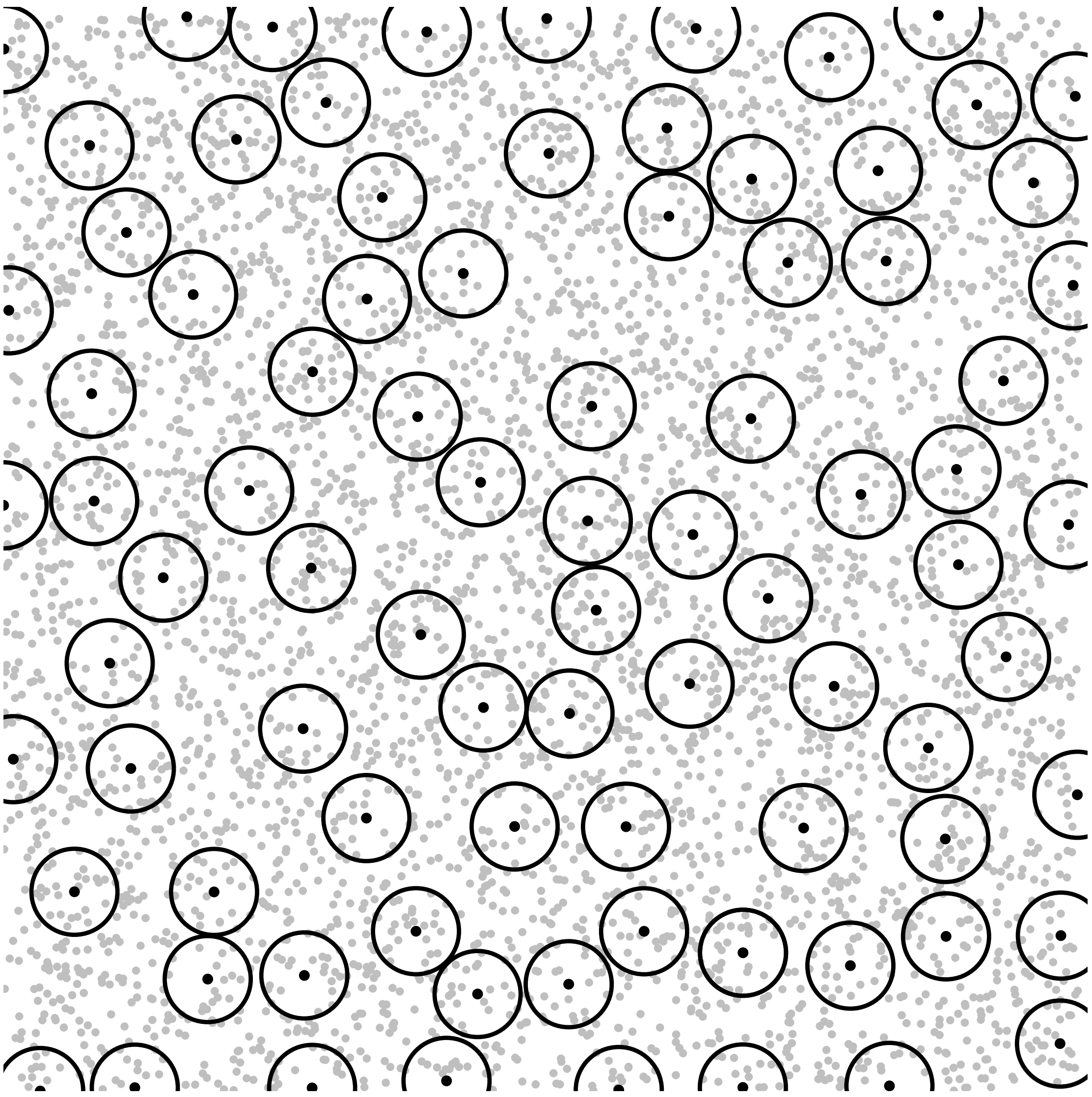
Figure 3.11: We applied two of our heuristics to a map[3.3] providing about 7,300 reference points of cities all over the world. We show the points in the background. We set $\delta = 1{,}600$ kilometers (or 1,000 miles.)

[4.3]downloaded from Natural Earth, `http://www.naturalearthdata.com/`, accessed Nov. 28, 2013



(a) shrinking-cones heuristic: 102 points, 8 seconds

Figure 3.12: We applied two of our algorithms to a synthetic data set where we randomly distributed 5,000 points in the unit square. We show unlabeled points in the background. We set $\delta = 0.08$.


(b) growing-cones heuristic, variant M1: 85 points, 18 minutes

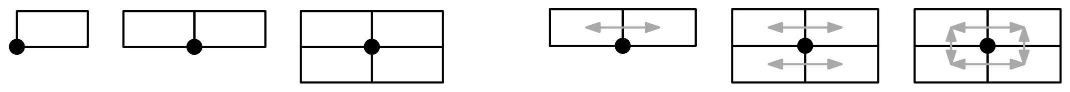
Chapter 4

Labeling Point Features with Sliding Labels

In this chapter, we examine another point-labeling problem. This time, we aim for dynamically attaching axis-parallel, rectangular labels to point features. In general, labeling point features requires a labeling model that defines possible label positions. Recall that there are two types of such models. In fixed-position models, each label is restricted to a discrete set of candidates relative to the point it labels; see Figure 4.1(a). In slider models [vKSW99], each label can be placed at any position such that (a certain part of) its boundary touches the corresponding point; thus, there is an unbounded number of candidates; see Figure 4.1(b). Usually, every point comes with a weight; the higher the weight the more important it is to label the point. We remark that the weight of a label $\ell(p)$ is the same as the weight $w(p)$ of its reference point p. Then, the aim is to maximize the sum of the weights of placed labels. This leads to the following static weighted point-labeling problem StatPointLab (for a fixed labeling model).

> Given a set P of points in the plane and, for each point $p \in P$, a weight $w(p)$ and a set $L(p)$ of label candidates, find a subset $\mathcal{L} \subseteq \bigcup_{p \in P} L(p)$ of label positions such that no two labels overlap, the sum $\sum_{\ell(p) \in \mathcal{L}} w(p)$ of the weights of placed labels is maximized, and $|L(p) \cap \mathcal{L}| \leq 1$ for each $p \in P$, that is, there is at most one label per point feature.

For fixed-position models in the static case, our problem is known as *maximum independent set* in weighted rectangle intersection graphs, which is known to be NP-hard [FPT81]. Moreover, the static problem is also known to be NP-hard for slider models [PSS$^+$03], even for the most restricted slider model, the *one-slider model*, where the bottom edge of the label must touch the corresponding point; see Figure 4.1(b).



(a) one-position model (1P), two-position model (2P), and four-position model (4P)

(b) one-slider model (1S), two-slider model (2S), and four-slider model (4S)

Figure 4.1: Examples of common labeling models: fixed-position models (left) and slider models (right).

Our Model. In this chapter, we are interested in a dynamic setting where the user can continuously pan and zoom a map in a 2D view. On that account, we consider a time interval $[0, T]$ in which the view is manipulated. We discretize this time interval into a sequence $t_1, t_2, \dots, t_h$, with $t_1 = 0 < t_2 < \cdots < t_h = T$, of points in time that correspond to frames. At any given time t_i, the user sees a rectangular region R_i of the map. This is the rectangle that originates if we project the screen on the map. When panning, the region R_i is translated on the map; when zooming, R_i is scaled.

We now can define the dynamic point-labeling problem DYNAPOINTLAB:

> For each $i = 1, \dots, h$, let $\mathcal{L}'_i$ be the set of labels in the view R_i that are placed at time t_i. We insist that all labels must lie completely within R_i. As in the static case, the quality of the current labeling is $W_i = \sum_{\ell(p) \in \mathcal{L}'_i} w(p)$. Then we define the overall quality of a dynamic label placement to be the quality, averaged over all frames: $\sum_{i=1}^{h} W_i / h$.

Note that DYNAPOINTLAB generalizes STATPOINTLAB, which corresponds to the restriction to a single frame ($h = 1$) and a large enough view R_1.

As we already know, there is a further requirement for interactive maps: the labeling should be consistent, that is, labels neither should jump nor flicker. To this end, Been et al. [BDY06] consider a continuous version of the objective function that we adopted in our model. They insist, however, that the position of a label relative to its reference point remains the same over all scales. We take a somewhat more pragmatic standpoint. We do allow labels to move. Still, our labels do not jump since we use the one-slider model and assume that our frame rates are high enough to ensure a smooth-looking movement when labels "slide". We do not, however, guarantee that labels do not flicker. We mitigate the problem for the user by introducing a simple *waiting list* that suppresses labels for about 30 frames (that is, between 0.5 and 4 seconds) after they disappeared.

With our algorithm we mainly target applications in which very large sets of points are to be labeled and thus time is critical, for example, the train radar for regional trains (RB/RE) of Deutsche Bahn[4.1] (German Railways) or browsers for large images of crowds that can be tagged[4.2].

Data Structure. We use a geometric data structure that allows us to efficiently predict collisions when pushing labels. In our application every label can slide only horizontally, that is, a label can collide only with labels to its left or right. We show how to use a *rectangulation* (see Figure 4.2) to access the relationships that matter and how to maintain the rectangulation when adding labels. A rectangulation is the special case of a trapezoidal map [dBCvKO08] where all trapezoids are rectangles. We obtain a rectangulation by shooting horizontal rays from the top and bottom edge of each label in both directions; a ray ends when it hits another label or the boundary of the view. A rectangulation consists of *labels* and *empty rectangles*. In order to ensure that each

[4.1] `http://bahn.de/zugradar`, accessed Feb. 6, 2014
[4.2] `http://www.u2.com/gigapixelfancam/`, accessed Feb. 7, 2014

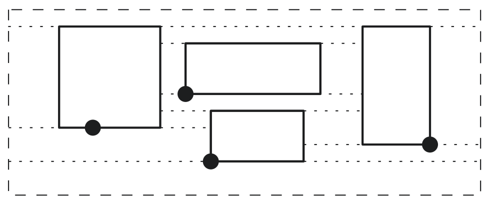


Figure 4.2: A rectangulation of a set of labels within a bounding rectangle R (the view) is a subdivision of R into labels and empty rectangles.

rectangle has a unique left and right neighbor, we add an empty rectangle of width zero between each two labels that touch horizontally. Note that a rectangulation corresponds to a visibility graph where labels are the nodes and empty rectangles are the edges; empty rectangles indicate which other labels a label "sees", that is, with which other labels a label might collide next if it slides. It is easy to see that, in our special case, the visibility graph is *planar* (broadly speaking, we can draw it in the plane without any crossing of edges). It is well-known that in planar graphs the number of edges is linear in the number of nodes (simplification of the Euler characteristic of planar graphs).

Our Contribution. We use the dynamic rectangulation data structure mentioned above to design an *online* algorithm for DynaPointLab, that is, we compute the labeling completely at runtime (see Section 4.2). Our heuristic proceeds incrementally. It repeatedly goes through all points in the view and tries to label each unlabeled point, one at a time. Our algorithm pushes away labels that have already been placed in order to make space for a new label. We suggest ways speeding up this algorithm for panning interactions (see Section 4.3). As we have implemented our approach, we present some experiments on real-world data (see Section 4.5). We conclude this chapter by discussing computation-time improvements for zooming operations, rotation operations, and dealing with a 3D view (see Section 4.6). A video that shows a result of our labeling approach can be found online[4.3].

4.1 Previous and Related Work

The problem of labeling maps with axis-parallel rectangular labels have also been studied from a theoretical point of view. As already stated, maximizing the number of labels in an overlap-free labeling using a fixed-position model is known to be NP-hard [FPT81]. Nevertheless, there are some approximation algorithms for the maximum-independent-set problem for the unweighted [AvKS98, CC09] and for the weighted case [EJS05, AW13]. More precisely, for the case of placing rectangles of arbitrary size, Agarwal et al. [AvKS98] present an approximation algorithm with factor $\mathcal{O}(\log n)$, where n is the number of rectangles, that runs in $\mathcal{O}(n \log n)$ time. The authors apply

[4.3] `http://lamut.informatik.uni-wuerzburg.de/dynapointlab.html` a divide-and-conquer approach. Further, for rectangles of unit-height, the authors give an approximation algorithm with factor 2 and a polynomial-time approximation scheme (PTAS) that are both based on dynamic programming. Chalermsook and Chuzhoy [CC09] improve the $\mathcal{O}(\log n)$-approximation of Agarwal et al. [AvKS98] to a $\mathcal{O}(\log\log n)$-approximation for axis-parallel rectangles of arbitrary size. They use *LP-rounding*, that is, they formulate and solve a linear program and round the fractional optimal solution to a non-optimal integer solution. Their algorithm is randomized in that it does not guarantee a feasible solution. The approximation factor, however, is always satisfied. For the weighted case, both Erlebach et al. [EJS05] and Adamaszek and Wiese [AW13] introduce a PTAS that is based on dynamic programming. Further, Erlebach et al. [EHJ$^+$10] present a PTAS for computing an overlap-free labeling that maximizes the sum of the weights of labeled point features whereas they use labels of equal height in a slider model. Recall that maximizing the number of placed labels in an overlap-free labeling using the slider model is NP-hard [PSS$^+$03], too.

For placing axis-parallel rectangular labels of unit-height to unweighted point features using a fixed-position model or a slider model, Van Kreveld et al. [vKSW99] give a an approximation algorithm with factor 2 that is based on a simple greedy strategy. By means of empirical tests on real-world data, the authors ascertain that a labeling using a slider model yields about 15% more labels than a labeling using the corresponding fixed-position model. Based on these results, we decided to use axis-parallel rectangular labels and the slider model in order to label interactive maps. Our tests show that we can improve the number of placed labels in the one-slider labeling model compared to the one-position labeling model even by 30–50%.

Goralski et al. [GGD07] present a geometric data structure for a problem that is similar to ours. Where we allow labels to slide horizontally, they consider vessels drifting in water. At any time, they need to know the neighbors of any vessel, that is, they need to know the potential counterparts for a collisions. The authors use a kinetic Voronoi diagram in order to predict the collisions. In our application, however, every vessel (that is, label) can slide only horizontally and thus can collide only with vessels to its left or right. Therefore, a Voronoi diagram does not reflect the adjacency relationship that is relevant in our application. On that account, we use a simpler data structure; we use a rectangulation.

For interactive maps, Harrie et al. [HSKL05] and Zhang and Harrie [ZH06] present real-time algorithms in order rule out positions where labels obscure other map objects. Their algorithms allow for an unbounded number of candidates but not for a dynamic labeling with labels that slide at runtime. It would be interesting to extend our algorithm by these approaches.

When a user interacts with an interactive map, the labeling has to be updated frequently. A naive approach is to perform each update by running a map labeling algorithm for static maps, not regarding the labeling that was visible before the update. Due to the recomputation of the labeling in each frame, however, labels flicker. Maass and

Döllner [MD06] present such an algorithm neglecting the labeling history. It attaches billboards with varying leader lengths to point features in real-time. Mote [Mot07] introduce an algorithm for labeling point features in interactive maps using the four-position labeling model. The algorithm requires labels of uniform size. With a little workaround and loss of quality, the algorithm can also deal with labels of arbitrary size. The author shows that his algorithm labels 5,000 points in 50 milliseconds and 75,000 points in less than a second. For this reason, he recomputes the labeling in each frame. Further, Luboschik et al. [LSC08] give a heuristic for the problem of maximizing the number of placed labels in an overlap-free labeling. They use the four-slider labeling model and, simultaneously, distant labels with leaders. According to their experiments, their approach is fully real-time capable although it computes the labeling in each frame. Due to the (additional) use of leaders, they often manage to label all points within the view. They do not, however, ensure that the leaders are crossing-free. This makes it hard to quickly decipher the labeling.

In order to support a consistent labeling when a user interacts with the view, approaches based on active ranges are frequently used (recall Chapter 3). Been et al. [BDY06, BNPW10] consider the problem of maximizing the total length of the active ranges for zooming operations while the position of a label relative to its reference point remains the same over all scales. Gemsa et al. [GNR11b] consider active ranges over scales when labels are allowed to slide horizontally and the points are restricted to lie on the x-axis. Gemsa et al. extend the idea of active ranges of scales to active ranges of rotation angles [GNR11a] and active ranges of time [GNN13].

As stated before, we doubt that the problem of labeling interactive maps can be solved with the help of precomputed active ranges alone since current digital maps allow for zooming, rotating, panning, and tilting operations. On the other hand, current algorithms that do not apply a precomputed data structure accept labels that flicker. Our approach with sliding labels, a waiting list, and a geometric data structure in the background can be seen as a compromise between these two worlds.

4.2 Incremental Algorithm

In interactive maps, new labels can appear whenever the user manipulates the view. To avoid that labels flicker, we build and maintain our labeling and the corresponding rectangulation *incrementally*. Additionally, we use a waiting list (see Section 4.3.1). One incremental step roughly works as presented in Algorithm 4.1: first, we locate the point to be labeled in the rectangulation. Next, we try to place its label such that it does not overlap other labels. This may imply that some labels have to be pushed away or to be removed. If the cost (in terms of summed weights) for these operations is too high, we do not execute them and instead reject the new label. Otherwise we update the rectangulation accordingly. In the remainder, we go through each of these steps in more detail.

Algorithm 4.1: IncrementalAlgorithm(P)

```
Input: set $P$ of points to be labeled
foreach point $p \in P$ to be labeled do
    determine the rectangle in the rectangulation that contains $p$ in order to
    quickly find elements that are involved when placing the label $\ell(p)$ of $p$
    slide $\ell(p)$ from its leftmost to its rightmost position and record the weights of
    the labels that have to be removed for keeping the labeling overlap-free
    slide $\ell(p)$ from its rightmost to its leftmost position and record the weights of
    the labels that have to be removed for keeping the labeling overlap-free
    combine the two sliding directions in order to determine a good
    position $\ell^*(p)$ for $\ell(p)$
    if placing the label of p increases the total weight of the labeling then
        place $\ell(p)$ at $\ell^*(p)$
        fix the rectangulation
```

4.2.1 Point Location

In computational geometry, point location in subdivisions is a well-known and well-solved problem. For trapezoidal maps, point-location data structures with logarithmic query time exist [dBCvKO08]. Since we did not want to invest too much time into implementing such a data structure without knowing whether point location was the bottleneck in our algorithm, we settled for a much simpler (though slower) approach.

Our search algorithm is a type of target-oriented breadth-first search; see Figure 4.3. Let p be the reference point to be labeled and let $y(p)$ be the y-coordinate of p; we define $x(p)$ accordingly. We start the search at the top left corner of the map. The left boundary of the map corresponds to a list of empty rectangles that is ordered by y-coordinate. We go through this list until we find the rectangle r whose y-interval contains $y(p)$. Then we test whether r contains p. If so, we are done. Otherwise, we go right. As each rectangle knows its unique right neighbor label ℓ, we can easily test whether ℓ contains p. If not, we continue the search from ℓ in the same manner as searching from the left boundary of the map until we find the element that contains p. Under the assumption that our rectangulation is roughly grid-like, the query time is $\mathcal{O}(\sqrt{n})$, where n is the current number of labels in the view; otherwise, due to the correspondence of the rectangulation and a planar visibility graph and as no label or rectangle is tested multiple times, the worst-case running time is $\mathcal{O}(n)$.

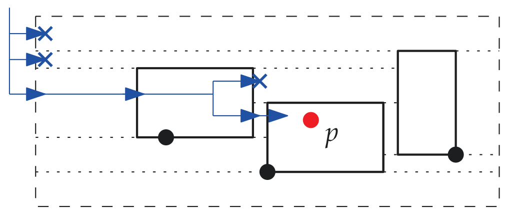


Figure 4.3: Illustration of the point-location algorithm. The point p is the reference point of the label to be placed.

4.2.2 Sliding Labels

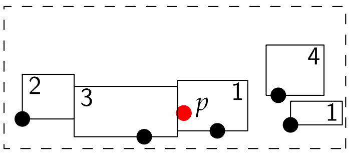
With the help of the point-location algorithm, we know the element of the rectangulation that contains the point p to be labeled. We next determine the final label position $\ell^*(p)$ of the label $\ell(p)$. In order to save computation time, we only label the current view. We require that labels rather vanish than overlap the view boundary. Normally, we have to make space for placing $\ell(p)$ by sliding and removing labels. Thus, we search for a position such that the sum of the weights of all removed labels is as small as possible (recall the weight of a label $\ell(p)$ is the same as the weight $w(p)$ of its point p). We first compute labelings at which labels can only slide to the left or to the right. We use the rectangulation to quickly query potential collision counterparts. While sliding, chains of labels (or *clusters*) form. Usually, there will be a label that finally prevents that we move the entire label chain further. Out of this chain, we remove a label that touches the view boundary or has reached its uttermost position and that has the lowest weight among those. At last, we compute a labeling at which labels slide in both directions by combining the two sliding directions. In the following, we describe this algorithm in more detail. For a better understanding, see Algorithm 4.2 and Figure 4.4. Only the final decision is visible to the user.

First, we set the label $\ell(p)$ to its leftmost position. We ignore all labels whose reference points lie to the left of p (we will correct this error by combining the two sliding directions). Next, we determine clusters of labels. To this end, we use a directed *contact graph* whose vertices are the labels that are currently visible. There is an edge between the vertex $v(p)$ of $\ell(p)$ and each vertex whose corresponding label overlaps $\ell(p)$ as well as between two vertices if the boundaries of their corresponding labels touch sideways. We direct an edge $(v(u), v(w))$ such that $x(u) < x(w)$. Finally, a cluster $c(s)$ is the set of vertices that can be reached by a (source) vertex $v(s)$; see Figure 4.5.

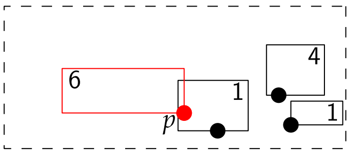
Assume that $\ell(p)$ is overlapped [Algorithm 4.2, line 4–13; Figure 4.4(b) and (c)]. By removing $\ell(p)$ from the contact graph, we obtain vertices without ingoing edges. Let $\ell(s)$ be such a vertex so that $\ell(s)$ additionally overlaps $\ell(p)$. We now slide the cluster $c(s)$ until it does not overlap $\ell(p)$ anymore, it touches another label, it touches the view boundary, or one of its labels reaches its rightmost position. We repeat rebuilding the conflict graph and building and sliding clusters until $\ell(p)$ is overlap-free or there is no cluster that we can slide further. If $\ell(p)$ is still overlapped, we determine a label $\ell(q)$ with a lowest weight that lies in the contact graph between $\ell(p)$ (excluding) and a

Algorithm 4.2 (sketch): SlidingToTheRight(p)

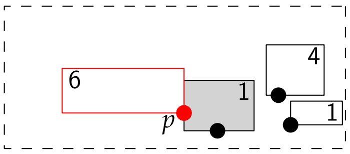
```
     Input: point $p$ to be labeled
     set the label $\ell(p)$ of $p$ to its leftmost position
     build the contact graph $G$
  FREE THE LABEL:
(4)  while $\ell(p)$ is overlapped and sliding an overlapping cluster is possible do
         update $G$
         $G' \leftarrow$ remove the vertex $v(p)$ (representing $\ell(p)$) from $G$
         determine a vertex $v(s)$ such that $v(s)$ has no ingoing edge in $G'$ and $\ell(s)$
         overlaps $\ell(p)$ in order to build the cluster $c(s)$
         slide $c(s)$ until $\ell(p)$ is not overlapped by $\ell(s)$ any longer; or a label within
         $c(s)$ reaches its rightmost position or touches the boundary
     if $\ell(p)$ is still overlapped then
         remove a blocking label and record the cost or reject the placement of $\ell(p)$
         slide labels that were slided by the blocking label back
         go to FREE THE LABEL
(13) // $\ell(p)$ is overlap-free
  SLIDE LABELS:
     update G
     build cluster $c(p)$
(17) while $\ell(p)$ has not reached its rightmost position and sliding $c(p)$ is possible do
         slide $c(p)$ until a label contained in $c(p)$ reaches its rightmost position or
         touches the view boundary
         update G
         build cluster $c(p)$
     if $\ell(p)$ has not reached its rightmost position then
         remove a blocking label or stop sliding to the right
         slide labels that were slided by the blocking label back
         update cost function
         go to SLIDE LABELS
(26) // $\ell(p)$ reached its rightmost position
```



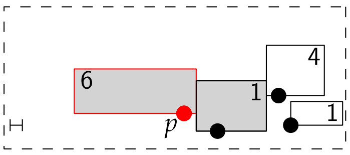
(a) the point p to be labeled appears



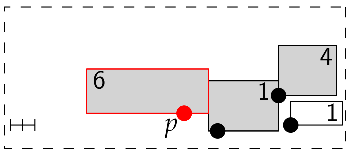
(b) set $\ell(p)$ to its leftmost position; neglect labels with reference points left to p



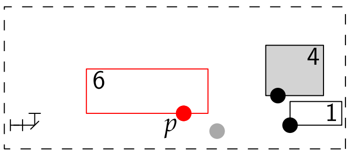
(c) slide overlapping label to the right to make $\ell(p)$ overlap-free



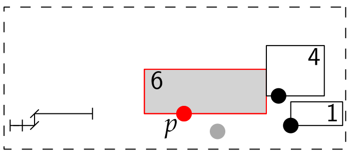
(d) $\ell(p)$ is overlap-free; slide cluster; record amplitude (left)



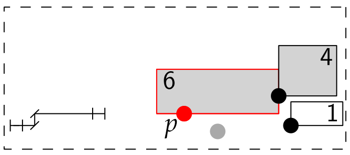
(e) slide cluster; record amplitude; blockade by rightmost position



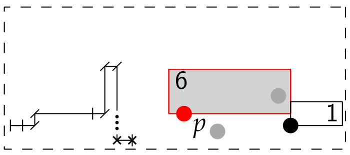
(f) remove cheapest label from cluster; record cost; some labels slide back



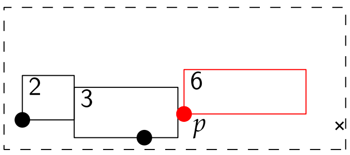
(g) slide further



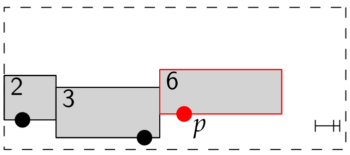
(h) slide further; next, raise blockade and slide further



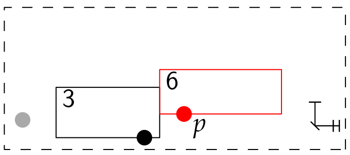
(i) set cost function negative due to weights; we are done



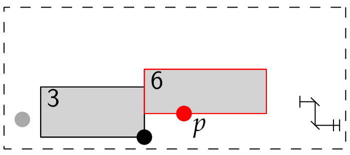
(j) from right to left: set $\ell(p)$ to its rightmost position



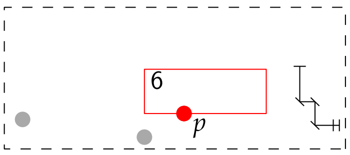
(k) slide; blockade by view boundary



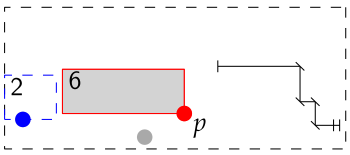
(l) raise blockade



(m) slide further; blockade by uttermost position

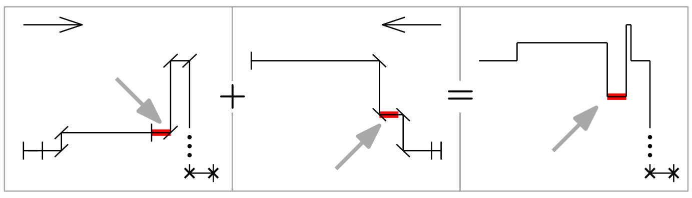


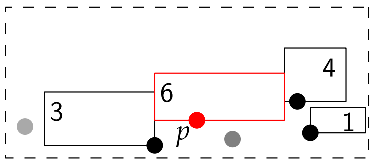
(n) raise blockade



(o) $\ell(p)$ reached leftmost position; *no* re-insertion of labels

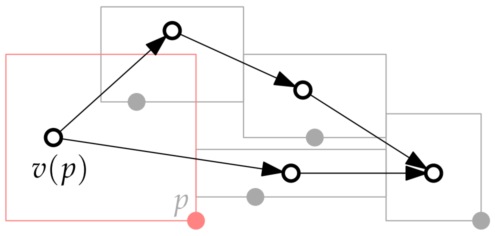
Figure 4.4: Illustration of several steps of the algorithm for sliding labels. The point to be labeled is p. We annotated every label with its weight. The rectangulation is not shown.


(p) cost functions, aggregated function, and minimum costs (arrowed)

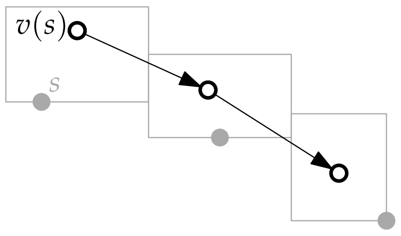


(q) a feasible final configuration

Figure 4.4: Final decision.



(a) a labeling and its corresponding contact graph



(b) a cluster $c(s)$ with source $v(s)$

Figure 4.5: A contact graph and one possible cluster.

blocking label (including), that is, a label that we cannot slide further as it has reached its rightmost position or as it touches the view boundary. If the weight $w(p)$ of p is too small, that is, if $w(p) \leq \sum_{d \in D} w(d) + w(q)$, where D is the set of removed (deleted) labels, we reject $\ell(p)$; otherwise we remove $\ell(q)$. Then, labels that were clustered with $\ell(q)$ and whose reference points lie to the right of q slide back until they touch another label or reach the position they had before they were slided by $\ell(q)$. We repeat building and sliding clusters and removing blocking labels until $\ell(p)$ is overlap-free.

As soon as $\ell(p)$ is overlap-free, we repeat the entire process with the objective that $\ell(p)$ reaches its rightmost position, that is, we slide $\ell(p)$ together with its cluster $c(p)$ [Algorithm 4.2, line 17–26; Figure 4.4(d)–(i)]. To this end, we modify the process as follows: we use $c(p)$ instead of $c(s)$; we use a cost function and stop sliding to the right instead of rejecting $\ell(p)$ due to weights. Whenever we remove a label $\ell(q)$, we store the weight of q and the current position of $\ell(p)$ at p, this is, the *amplitude*, in a cost function [Figure 4.4(d) et seqq.]. If $w(p) \leq \sum_{d \in D} w(d) + w(q)$, we stop sliding $\ell(p)$ to the right and set the cost function from this amplitude to the rightmost position to $-\infty$ [Figure 4.4(i)].

With the costs and amplitudes that we have stored, we finally obtain a step function for sliding $\ell(p)$ to the right. We repeat the entire process for sliding $\ell(p)$ from its rightmost to its leftmost position [Figure 4.4(j)–4.4(o)]. We sum up the cost functions [Figure 4.4(p)]. These aggregated costs represent the cost for sliding some labels to the right and some to the left. Next, we extract the minimum of the aggregated function.

Note that the minimum (normally) is bounded by two amplitudes. Indeed, each label position for $\ell^*(p)$ between these two amplitudes yields the same cost. There are several criteria to decide for one position. In our implementation, we choose a low-cost amplitude that causes the fewest labels to slide. Now, we make our final decision visible for the user. To this end, with the help of the cost function, we once more slide some labels to the right and some to the left—this time simultaneously—in order to make space to place $\ell^*(p)$.

Note that our algorithm is a heuristic. In Figure 4.4(o) we could re-insert the label on the left. So, we sometimes overestimate the total cost. This can result in the choice of a non-optimal amplitude, this is, we place fewer labels than possible. If we (try to) label unlabeled points in each frame, this error is quickly fixed.

In some unfavorable cases, the algorithm for sliding labels has an asymptotic running time of $\mathcal{O}(n^2)$, where n is the number of labels within the view; see Figure 4.6. In such an unfavorable case, we remove all labels from the rectangulation whereas we slide labels $n \cdot (n+1)/2$ times. Nevertheless, these cases are rather unlikely for real-world data. Note that, after sliding, the update of the rectangulation, only needs constant time. In the next paragraph, we will see that, when removing a label from the rectangulation, we need linear time in the worst case. Hence, the update does not need more than $\mathcal{O}(n^2)$ time.

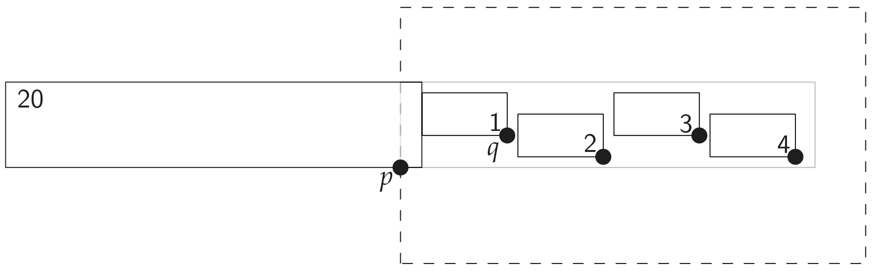


Figure 4.6: The gray rectangles shows $\ell(p)$ at its rightmost position; numbers indicate weights. After placing the label $\ell(p)$ to be placed at its leftmost position, we slide all n labels. Then, we remove $\ell(q)$. The remaining labels slide back. Next, we slide $n-1$ labels, and so on.

4.2.3 Fixing the Data Structure

We now discuss how to update the rectangulation after sliding, removing, or placing a label.

Sliding a label $\ell(q)$ is the easiest operation since it does not change the topology of the rectangulation. We only have to update the widths of the empty rectangles to the left and right of $\ell(q)$, their amplitudes, as well as the amplitude of $\ell(q)$.

Removing a label $\ell(q)$, however, is slightly more complicated; see Figure 4.7. By means of the rectangulation, we directly know all the left and right neighbor rectangles of $\ell(q)$. To find the neighbor rectangles above and below $\ell(q)$, we perform a search, originating from $\ell(q)$, that is similar to the point-location algorithm; see Figure 4.7(a). We start at the bottommost left neighbor rectangle of $\ell(q)$. We move repeatedly from rectangle to label until we find a label $\ell(r)$ whose corresponding y-coordinate $y(r)$ is smaller than $y(q)$. Now, we move back to the right until we find the rectangle that touches $\ell(q)$ from below. We repeat this to also find the upper rectangle. Finally, the set of neighbors of $\ell(q)$ is complete; see Figure 4.7(b). We remove $\ell(q)$ and extend the horizontal edges of its neighbor rectangles to close the gap left by $\ell(q)$; see Figure 4.7(c). As the number of empty rectangles influences the computation time deeply, we finally merge rectangles that are vertically adjacent to each other and have the same left and right neighbor.

We add a new label $\ell^*(p)$ to the rectangulation after we have eliminated and slided existing labels to make space for $\ell^*(p)$. Therefore, we must not care about label–label overlaps. Still, we need to update the rectangulation. For this purpose, we first detect all empty rectangles that $\ell^*(p)$ overlaps. Again, we use a search similar to the point-location algorithm; see Figure 4.8. Starting from the rectangle r that contains p we go to the left neighbor of r. Now, we repeatedly move from the topmost left neighbor rectangle to the next label until we reach a label whose top edge lies at a higher y-coordinate than the top edge of $\ell^*(p)$. From every label we passed while going left, we start to go right. We stop if we find a rectangle that lies completely above or below $\ell^*(p)$, that overlaps $\ell^*(p)$, or that we have visited before. During this search we collect all rectangles that overlap $\ell^*(p)$. Next, we split each of these rectangles into at most three new rectangles, that is, the part above $\ell^*(p)$, the part below $\ell^*(p)$, and the remaining middle part. This middle part again needs to be split into at most three parts, that is, the part left of $\ell^*(p)$, the part right of $\ell^*(p)$, and the part covered by $\ell^*(p)$. After splitting $\ell^*(p)$ into its parts (see Figure 4.9 for the result) we need to merge rectangles that are vertically adjacent to each other and have the same left and right neighbor.

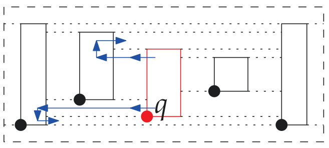
It is easy to see that, due to the correspondence of our rectangulation and a planar visibility graph, no steps needs more than $\mathcal{O}(n)$ time.

4.3 Running Time Improvements

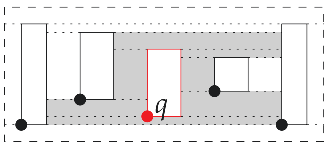
The incremental algorithm is quite fast. Triggering it in each frame for testing if we can place a new label or updating label sizes due to zooming operations is time consuming, though. Therefore, we present two concepts to speed up the algorithm. First, we introduce a waiting list; this is, we wait several frames until we try to label a certain reference point again. Furthermore, for panning operations, we discuss how to predict the point in time at which we have to trigger an update of the rectangulation.

4.3.1 Waiting List

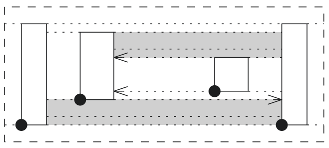
Certainly, in a view, there can be many reference points without labels. It is rather unlikely that, in the current frame, we can place a label that we could not place in the



(a) search rectangles above and below $\ell(q)$



(b) rectangles to update are shaded



(c) lengthen lines; rectangles to merge are shaded

Figure 4.7: Illustration of several steps of the algorithm for updating the rectangulation. The label to be removed is $\ell(q)$.

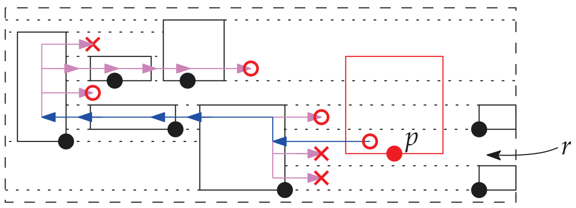
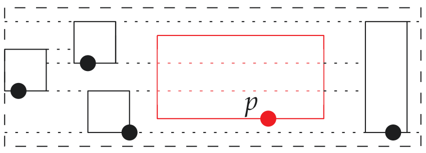
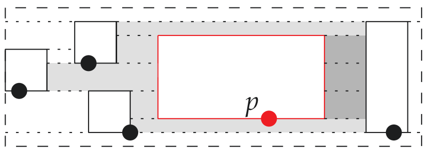


Figure 4.8: Illustration of the search originating from the rectangle r that contains the point p to be labeled for detecting rectangles overlapped by the label $\ell^*(p)$. A circle indicates an overlap with $\ell^*(p)$, a cross indicates the end of a search path.



(a) situation before placing $\ell^*(p)$



(b) shaded rectangles were built, dark shaded rectangles must be merged

Figure 4.9: How to update the rectangulation if we place the label $\ell^*(p)$.

preceding frame. Additionally, it does not disturb the user if we place a label with a small delay. Due to these considerations, we introduced a waiting list.

We always try to label all reference points that just appeared. Let p be a reference point that we unsuccessfully tested for placing its label. On that account, we add p to the list W of waiting reference points. Now, we wait at least for f frames until we test p again. (We only count frames with interactions, though.) For load balancing, we just test a certain number M of labels. Currently, M is the minimum of $|W|/f$ and all labels whose last test lies at least f frames in the past. Thereby $|W|$ is the number of labels; in W; $|W|/f$ is an empirical value.

Recall that the algorithm for sliding labels does not re-insert labels; see Figure 4.4(o). If we use the waiting list, it lasts some frames until a label appears again. Sometimes, it also can happen that an unimportant label instead of an important one is placed awhile. Moreover, the waiting list can cause quickly changing labels. Consider a labeling. In frame f_i, we place the label $\ell(p)$. In f_{i+1} a more important label makes $\ell(p)$ disappear. When we continue this, each of such labels is only visible for a single frame. There are several possibilities to solve this problem. We could state that we must not remove a just-placed label $\ell(p)$ for several frames. We also could increase the weight of $\ell(p)$ and decrease it little by little. The latter approach has the advantage that we place labels with a much higher weight than $\ell(p)$ earlier than labels that are only slightly more important than $\ell(p)$.

4.3.2 Predicting Changes of the Rectangulation for Panning Operations

When a user pans or zooms, we need to update the rectangulation. As we have not implemented the prediction for zooming operations, and thus the concept is not verified yet, we give our ideas in Section 4.6.

For panning operations, it is, however, easy to predict the *event points* at which changes will be necessary. When panning, labels in the map will not intersect unless a new label appears at the view boundary or a label is blocked by the view boundary and thus needs to slide. This allows us to compute the distance that the user can pan to the right, left, bottom, and top without any event. If a reference point enters the view, we can apply the incremental algorithm of Section 4.2 in just the same way as for any other point feature. In the case that a label touches the view boundary, we can treat the boundary as a big label that must not be moved. Thus, the touching label slides (or finally vanishes) rather than it crosses the boundary. We can apply the algorithm for sliding labels of Section 4.2. Whenever a new label was added or a label was slided, we compute new event points; this needs $\mathcal{O}(n)$ time.

4.4 Putting Things Together

After introducing all parts of our algorithm for labeling point features with sliding labels, we put the single concepts in Algorithm 4.3 together. We call the algorithm in each frame with an interaction. Note that the algorithm idles if no label intersects the view boundary and no reference point is tried to be labeled.

For panning interactions, our algorithm has an asymptotic running time of $\mathcal{O}(n^2)$ in each frame in that an event occurs or a waiting reference point is tried to be labeled. While zooming, we recompute the labeling in each frame. Our algorithm has a worst-case running time of $\mathcal{O}(n^3)$ for each frame (for example, when, similar as shown in Figure 4.6, $n/2$ labels are placed within the view and $n/2$ labels are placed at the view boundary). Nevertheless, it is a reasonable assumption that, especially on real-world data, it is rather unusual that our algorithm needs a quadratic or even cubic number of steps. Thus, our algorithm is still better than a labeling algorithm using a naive collision detection that *always* needs a quadratic number of steps when it tests for each label if it overlaps any other label. If we permit sliding in the naive approach, the worst-case running time is $\mathcal{O}(n^4)$.

Algorithm 4.3: LabelingAlgorithm( )

```
if the user pans and an event occurs then
    update the positions of the labels within the rectangulation
    foreach label $\ell(p)$ that intersects the left view boundary do
        try to slide $\ell(p)$ to the right // analog to Algorithm 4.2
    foreach label $\ell(p)$ that intersects the right view boundary do
        try to slide $\ell(p)$ to the left // analog to Algorithm 4.2
if the user zooms then
    increase the weight of each labeled point suitable
    compute the rectangulation for the current point set from scratch
      // see Algorithm 4.1
    reset the increased weights
apply the incremental algorithm for each point that entered the view
      // see Algorithm 4.1
apply the incremental algorithm for each point out of the waiting list that is
already permitted to be tested // see Algorithm 4.1 and Section 4.3.1
append unlabeled points to the waiting list
if a label was slided or a label was inserted then
    compute the next event points // see Section 4.3.2
```

4.5 Experiments

We have implemented the incremental algorithm of Section 4.2 using a rectangulation and the waiting list of Section 4.3.1. For zooming operations, we recompute the rectangulation in each frame. In order to estimate the usefulness of our algorithm, we compare it to a naive approach. The naive approach differs from the rectangulation-based approach in how it detects overlapping labels and potential collision counterparts. Instead of using a geometric data structure, the naive approach repeatedly checks *all* pairs of visible labels. The naive approach yields the same labeling as the rectangulation-based approach.

Both approaches have in common that we can (i) use the waiting list and (ii) replace the slider model by a fixed-position model where the center of the label's bottom edge touches the reference point.

For our implementations, we used C++ with OpenSceneGraph 3.0[4.4]. We executed our experiments on a Windows 7 system with a 3.3-GHz AMD triple-core processor, a GeForce GTX 460 graphics card, and 8 GB of RAM, applying the Microsoft Visual Studio 2010 Ultimate compiler in 32-bit release mode. The complete code has about 12,300 lines. For our tests, we used a world map from Natural Earth[4.5] providing 7,322 cities as weighted points; see Figure 4.10. We scaled the weights such that unimportant points had weight 1; important points had weight 4. We implemented several different single-interaction camera paths, this is, paths for only panning and for only zooming. Each of these paths takes 24 seconds whereas we never pause an interaction (as idling would increase the frame rate). Additionally, we defined multi-interaction paths where the view is manipulated by panning, zooming in, and zooming out operations. Each of these paths executes its interactions for 42 seconds. For all single and multi-interaction paths, on average, either 35, 105, or 205 labels are visible. For each of these numbers, we implemented three different paths. We taped one of the multi-interaction paths and, as mentioned at the end of the introduction of this chapter, we made the resulting video available online[4.6]. Figure 4.11 as well as Figures 4.19–4.21 at the end of this chapter show some screenshots of our program.

To the total 27 different paths, we applied the naive approach as well as the rectangulation-based approach with and without sliding and with and without a waiting list in that a point remains for at least $f = 30$ and $f = 60$ frames.

For determining the width of a rectangle, we counted the number of the letters in the city name and scaled it with an empirical value that depends on the desired width of a letter and with the weight of the label. As the drawing routine of OpenSceneGraph for Windows is rather time consuming, we "only" drew reference points and labels in the view. Without drawing any object but computing the rectangulation, we received frame rates more than 400 FPS.

[4.4]`http://www.openscenegraph.org/`, accessed Nov. 24, 2013
[4.5]`http://www.naturalearthdata.com/`, accessed Nov. 28, 2013
[4.6]`http://lamut.informatik.uni-wuerzburg.de/dynapointlab.html`

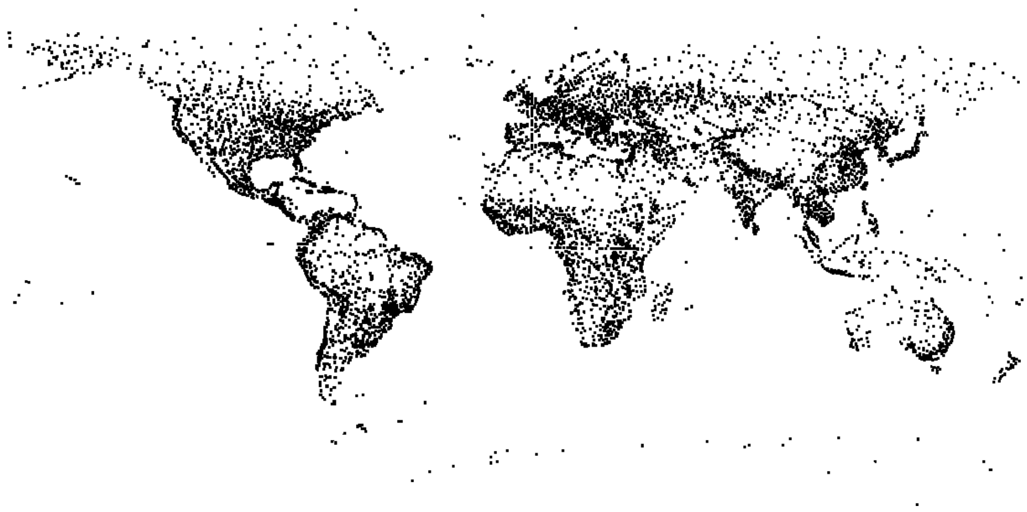
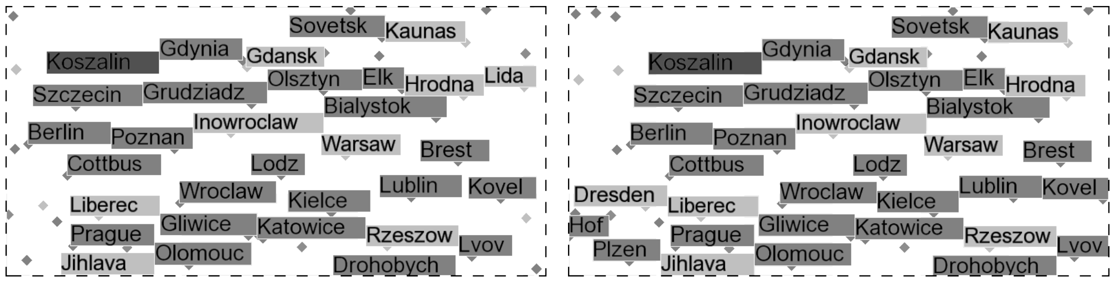
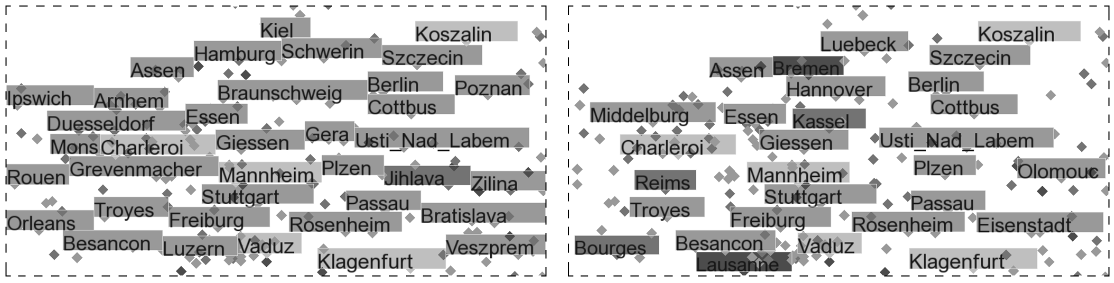


Figure 4.10: Point set of world map that we used in our experiments.



(a) From left to right: the user pans to the right; new labels appear at the left boundary of the view; at the right boundary, a label vanishes. On the lower right, Lvov pushed Rzeszow, Rzeszow pushed Katowice, and so on.



(b) A map that was labeled using the one-slider model (left) and the same map that was labeled using a one-position model (right). We immediately (visually) perceive that the number of the labels in the left figure is higher.

Figure 4.11: Screenshots of the implementation of our algorithm.

For each frame, we recorded the sum of weights of all labeled points. We summed up the weights of the three paths with the same interaction type and the same average number of labels in the view. Finally, we averaged the weights over the total number of frames in order to compute the average quality; see Table 4.1 (quality). Additionally, we averaged the total number of frames over the processing time in order to compute the frame rate in frames per second; see Table 4.1 (frame rate).

We observed that in many cases, the frame rate is rather low when we start a camera path as well as while zooming. Recall that, in these cases, we compute the rectangulation from scratch. We observed further that our algorithms yield different results with regard to the averaged weight and the frame rate for each pass of the *same* camera path. This is because the current load factor influences our measurements. As a result, also the average quality of our algorithm and the naive approach differ slightly. Since the difference is not noteworthy, Table 4.1 shows only the quality results for the rectangulation-based algorithm. As the results for zooming in only slightly differ from the results for zooming out (the inverted path), we only averaged the results for zooming out.

We conclude that, using the slider model, our algorithm yields an improvement of 30–50% in the labeling quality with respect to the algorithm using the fixed-position model. Second, we point out that the rectangulation-based approach increases the frame rate by up to 40% if the screen contains a large number of labels. If we additionally use a waiting list in that a point remains for at least 30 frames, the frame rate for small point sets increases by about 15%. For large point sets, it sometimes doubles. The maximum loss in quality due to the waiting list is 18%. When we apply a waiting list in that a point remains for at least 60 frames, to our surprise, the frame rates increase by at most 2 FPS whereas the quality drops by up to 30%. Therefore, we do not show the details for the waiting list suppressing labels for 60 frames in Table 4.1.

4.6 Extensions and Comments

We finally present some unverified concepts for the prediction of event points while zooming, handling rotation operations, and dealing with a 3D view.

Predicting Changes of the Rectangulation for Zooming Operations. While the user zooms, we require that each label keeps its size on the screen. Instead of considering the movement of points while the user zooms, we change the sight: on a map with constant size, each object (including the view) grows if the user zooms out; objects shrink, if the user zooms in. Certainly, while zooming, empty rectangles can collapse; see Figure 4.12. Moreover, the y-order of edges of placed labels can change as labels grow and shrink by a scale *factor*. This makes the prediction of event points and a local update while zooming difficult.

There are two types of events for zooming operations: *collapses of empty rectangles* and *emerges of empty rectangles*; see Figure 4.12. (A label–label overlap is no event as

		rectangulation			
quality		$f = 0$		$f = 30$	
	$\varnothing\lvert\mathcal{L}'\rvert$	1P	1S	1P	1S
pan	35	71	102	69	97
	105	201	302	196	277
	205	417	605	404	568
zoom	35	54	81	53	79
	105	178	259	162	225
	205	375	547	318	452
both	35	71	107	68	99
	105	197	294	183	258
	205	394	582	344	479

		naive				rectangulation			
frame rate		$f = 0$		$f = 30$		$f = 0$		$f = 30$	
	$\varnothing\lvert\mathcal{L}'\rvert$	1P	1S	1P	1S	1P	1S	1P	1S
pan	35	49	33	51	37	50	33	52	38
	105	13	8	14	10	18	11	19	14
	205	8	4	8	6	9	6	10	7
zoom	35	60	37	60	41	60	38	61	42
	105	19	12	21	15	19	12	21	16
	205	9	5	11	8	9	6	11	8
both	35	46	28	48	34	45	28	48	34
	105	17	10	18	13	17	11	19	14
	205	7	4	8	6	9	6	10	8

Table 4.1: Measured values for the quality (with respect to our objective function) of our labeling, averaged over frames, and the frame rate. We denote the number of frames that a point (at least) remains in the waiting list by f; $\varnothing\lvert\mathcal{L}'\rvert$ is the average number of labeled points on the screen; *1P* and *1S* are the used labeling models.

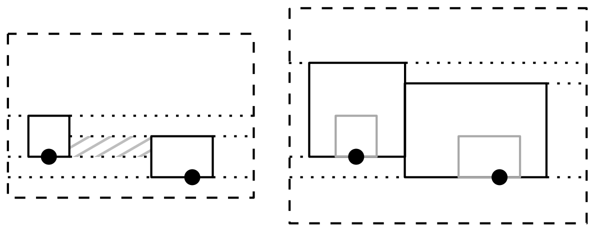


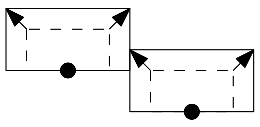
Figure 4.12: From left to right: if the user zooms out, the hatched rectangle collapses horizontally. From right to left: if the user zooms in, a new rectangle emerges.

this is always preceded by a collapsing rectangle.) A rectangle collapses if its height or width gets zero. An emerging rectangle means to insert a rectangle of height or width zero into the rectangulation; presumably, it will grow in the next frame. Now, for each rectangle, we test after which zooming distance the rectangle collapses and which are the involved labels. We directly know the horizontally-neighbored labels. We detect labels above and beneath a vertically-collapsed rectangle with the help of an algorithm similar to our point-location algorithm of Section 4.2.1. We compute and store the next event for zooming in and the next event for zooming out.

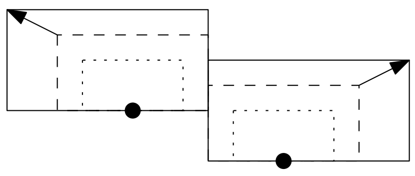
If an event occurs (or we want to label an unlabeled point), we first have to correct the sizes of all rectangles and labels as, due to the prediction of events, there is no need to correct the rectangulation in each frame. If a rectangle collapses, in the next step either a rectangle emerges or two labels will overlap. If a rectangle emerges, we add it. If two labels touch sideways, we simultaneously grow and slide labels; see Fig. 4.13. Sooner or later, we must remove a label: we choose the less important one and apply our algorithm for fixing the rectangulation of Section 4.2.3. In the whole process, we have carefully to make sure that the stored neighbors are correctly updated.

Handling Rotation Operations. Our current algorithm is not able to handle rotation operations. If the user rotates the view, labels move on a circle around the center of the view. Both the x- *and* y-coordinate of each reference point changes; see Figure 4.14. In order to update the rectangulation, for each reference point, we compute the change of the x- and y-coordinates; for each empty rectangle, we test if it collapsed. Without any exception handling, our current algorithm sometimes makes a wrong decision; see Figure 4.15.

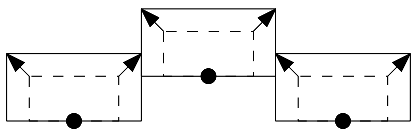
Nevertheless, assume that we can adapt our algorithm suitable (possibly, by also maintaining top and bottom neighbors or taking the movement of the reference points into account). Then, there is still the problem that we should update the rectangulation in each frame. We *can* predict after which rotation angle we have to update the rectangulation but as soon as the user manipulates the view with another interaction than a rotation operation, all precomputed values are outdated. On the other hand, there are many vertical collapses in a short interval of time. We expect that computing the rectangulation in each frame from scratch is faster. To this end, we recommend to complement our algorithms by a completely different approach for rotation operations—possibly



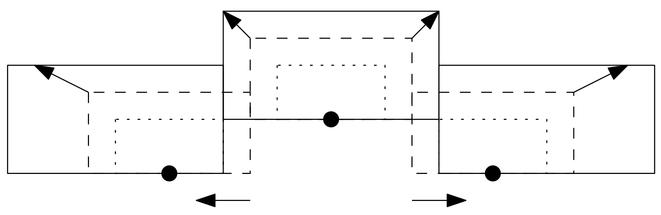
(a) labels grow equally until they touch



(b) labels change their growing directions



(c) labels grow equally until they touch



(d) labels change their growing directions and make other labels slide

Figure 4.13: Labels can grow and slide simultaneously. This corresponds to a change of the growing direction.

even without the support for sliding labels as, if the user rotates the view, labels might press from the left and the right simultaneously. Moreover, if the user rotates at least 180°, labels that before made other labels to slide in one direction now make the labels drifting to the opposite direction; see Figure 4.16.

We remark that, although there are similar problems for zooming and rotation operations, the predicted event points for zooming are longer valid than the predicted event points for rotation operations; that makes the usage of a rectangulation for zooming operations still worth a try.

Handling the 3D View. We observe that, in maps with a perspective view, labels move with different speeds; for example, if a label in the foreground of the view moves from the left view boundary to the right boundary, a label in the background that also started at the left view boundary, does not reach the right boundary. This makes the prediction of the movement of reference points even harder. Furthermore, two labels that do not overlap in *world space*, that is, on the input map, sometimes overlap in *screen space*, that is, the indeed displayed map; see Figure 4.17. For these two reasons, we project the reference points from world space to screen space in each frame (see Figure 4.18) and compute the rectangulation in screen space. In the the related-work section of Chapter 2, we stated that experts recommend to use smaller labels at the back and larger labels in the front of the view as this improves the spatial perception [MJD07b, VTW12]. With our current approach, this is not possible, though. We additionally learned that the size of a label should not be influenced by the weight of the label. Thus, we recommend to define a *standard* label size. We use this size in the rectangulation and scale the labels

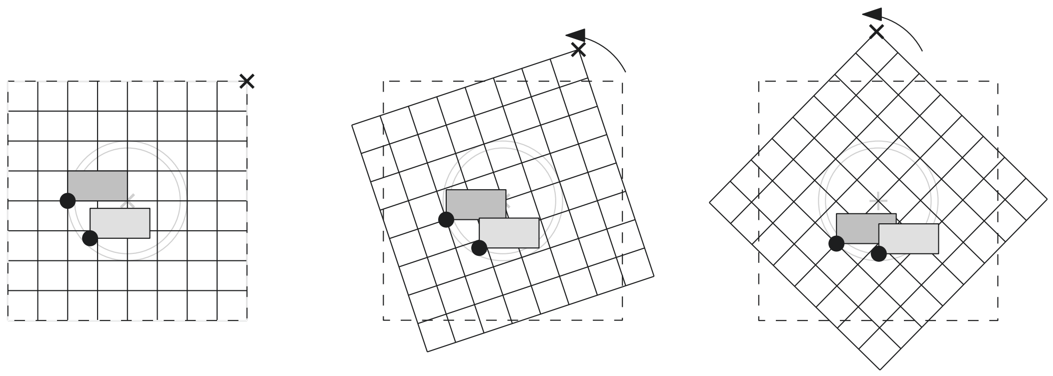
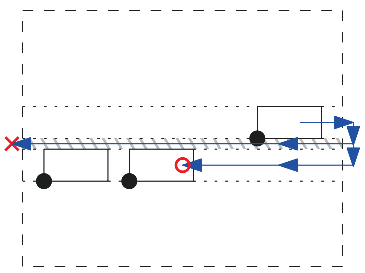
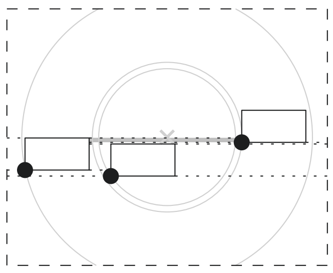


Figure 4.14: If the user rotates the view, coordinates of reference points change; labels might get overlapped.

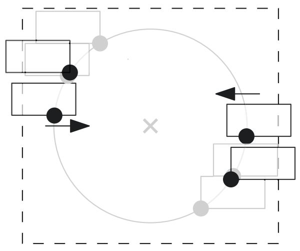


(a) searching for the new neighbor in the rectangulation before rotating; the hatched rectangle collapses at a rotation of 3°

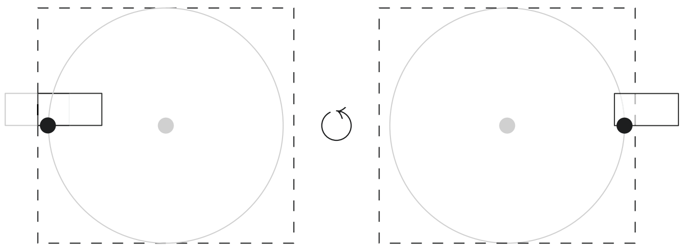


(b) correctly updated rectangulation after a rotation of 4°; the gray rectangle emphasizes the correct new neighbor

Figure 4.15: Without any exception handling, our algorithm makes a wrong decision.



(a) after a rotation of 25°, labels press from the left and the right simultaneously



(b) first the label pushes other labels to the right; after a rotation of 180°, the label pushes other labels back to the left

Figure 4.16: Unfavorable situations while rotating. Gray labels indicate former positions.

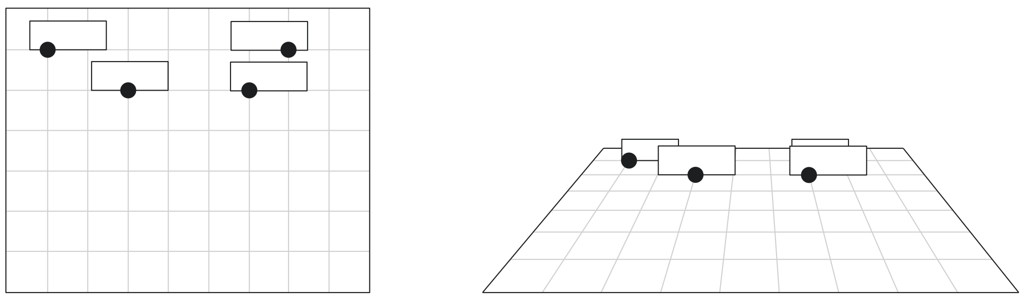


Figure 4.17: Labels that do not overlap in world space might overlap in screen space. Labels that are nearer to the observer are larger. If we lengthen the shown lines in the perspective view, they finally meet in the vanishing point.

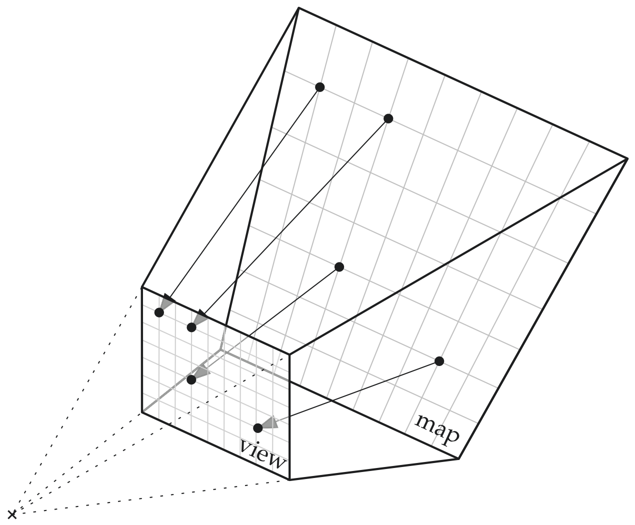


Figure 4.18: Projecting reference points from world to screen space. For the sake of simplicity,we project the *entire* world map into a *2D* view.

to be displayed. If we, for example, use the size of a label in the front of the view as standard size, we avoid any label–label overlap. If we use the size of a label in the middle of the view, labels in the front might overlap. We think that slightly overlapping labels are acceptable in a 3D view as they again should improve the spatial perception. Moreover, we observe that, in a 3D view, only the lower part of a label can be overlapped whereas the upper part of a word is more important to decipher the word [Bac05] (see related-work section of Chapter 2). In both the proposed choices for the standard size, at the back of the view, there are probably less labels than possible. In our opinion, this is also reasonable as labels at the back are less important.

If the user *pans horizontally*, labels in the foreground move faster than labels in the background. In order to predict the next event point, we first compute an event point as normal and then we scale the value such that the scaled value indicates how much the center of the view can move until the event occurs. If the user *pans vertically*, labels that move to the front, grow; labels that go to the back, shrink. One the one hand, we must scale the labels for the visualization suitable; one the other hand, we must pay attention to empty rectangles that collapse horizontally as the reference points move along the lines that head to the vanishing point. We have to consider this for the computation of the event points. Further, we assume that a label can grow or shrink because of the movement within the map or shrink at the back due to a *tilting* operation but not by a *zooming* operation. On that account and as we fixed the label size used in the rectangulation, we can handle both these interaction in the same way as panning vertically.

4.7 Concluding Remarks

In this chapter, we have described an algorithm that dynamically labels points in interactive maps using a slider model. To speed up our algorithm, we used a rectangulation data structure and a waiting list. We conclude that sliding labels improve the labeling quality (in terms of our objective function) by up to 50%. Compared to a naive approach, our heuristic significantly improved the frame rate; in some cases, it even doubled.

For the future, it would be interesting to develop and implement an algorithm that solves our problem optimally. As we require that the labeling considers the history of the labeling, we cannot just optimize the output frame-wise. Another enhancement would be that we adapt our current heuristic such that is able to handle rotation operations and to deal with a 3D view. Further, there are several points that could improve the computation time of our implementation. We could analyze the computation time of our current simplistic point-location strategy. Will it be worthwhile replacing it with a dedicated dynamic point-location data structure? How much computation time can we save by predicting event points for zooming operations? In the related-work section of this chapter we stated that we did decide for using a rectangulation instead of a Voronoi diagram as data structure because labels can only slide horizontally. This is true but while zooming, labels change their heights what makes the update of the rectangulation quite sophisticated. Possibly, it is easier to handle zooming and rotation operations with a data structure that is based on a Voronoi diagram. A completely different approach is not to improve our algorithm but the naive approach. As experience teaches, approaches using a grid-based collision detection are quite fast.

Last but not least, it would be interesting to conduct a user study in order to learn how users cope with the additional cognitive load of sliding labels.

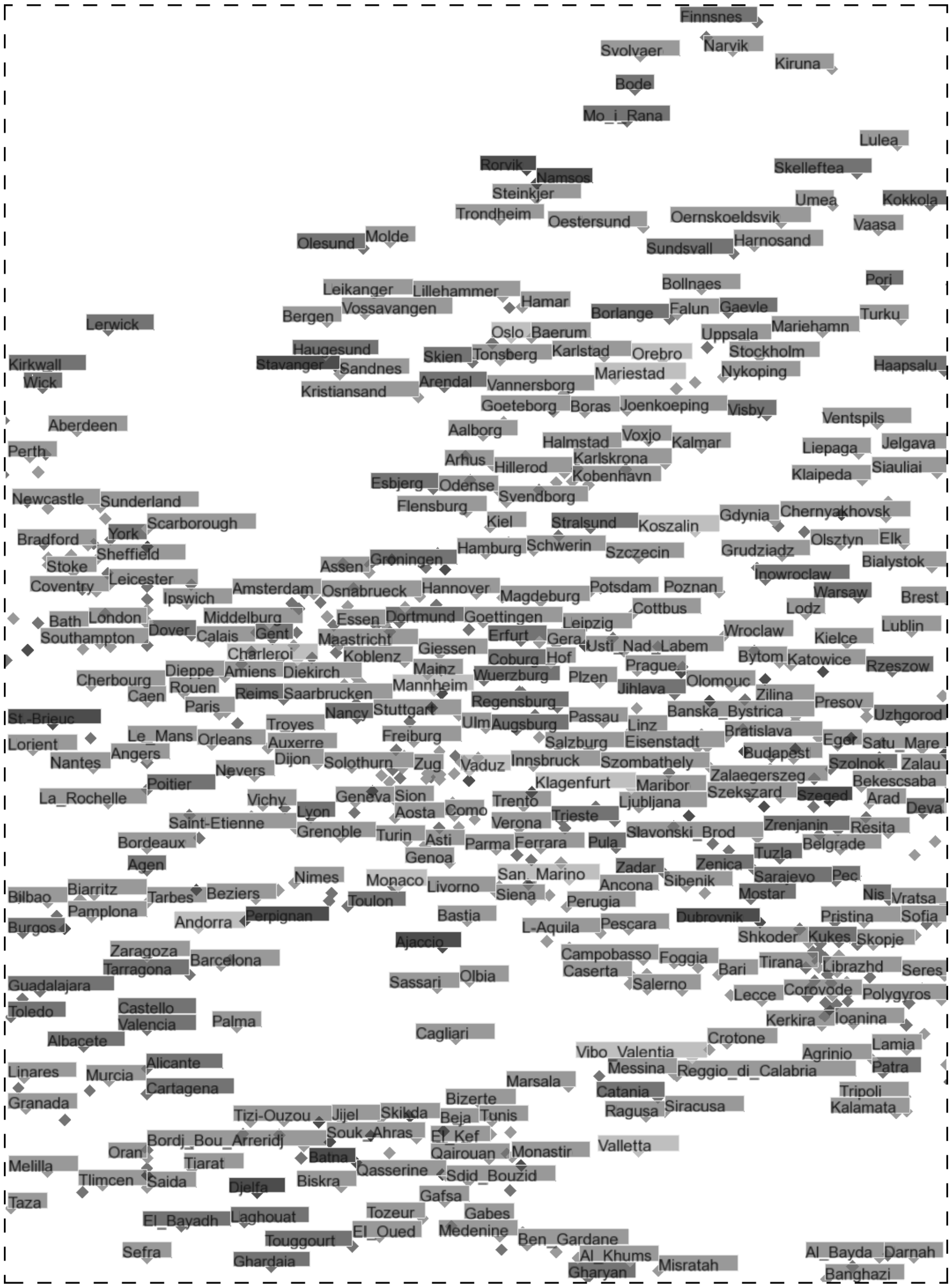


Figure 4.19: A map of Western Europe labeled with our algorithm.

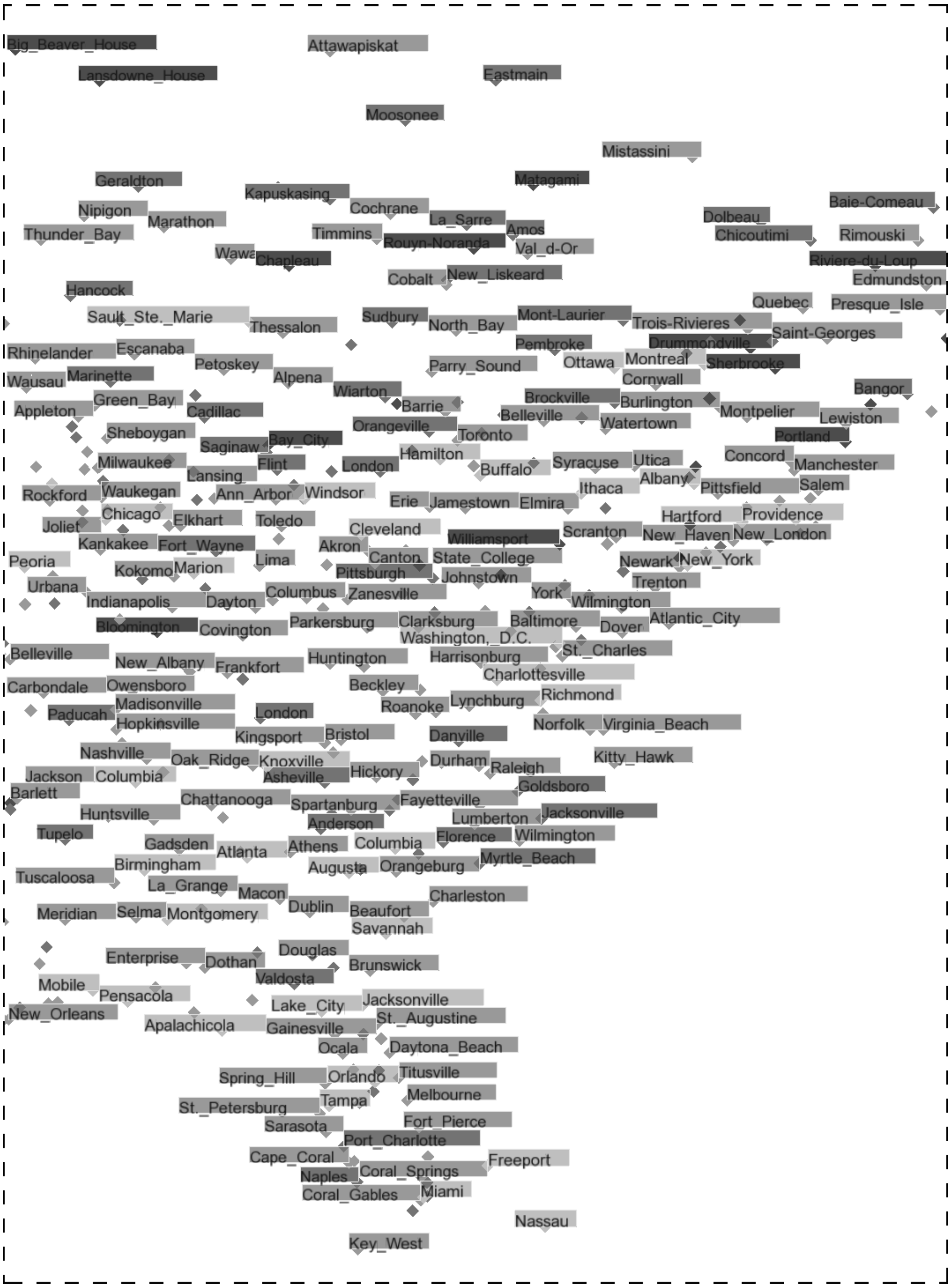


Figure 4.20: A map of the East Coast of the United States labeled with our algorithm.

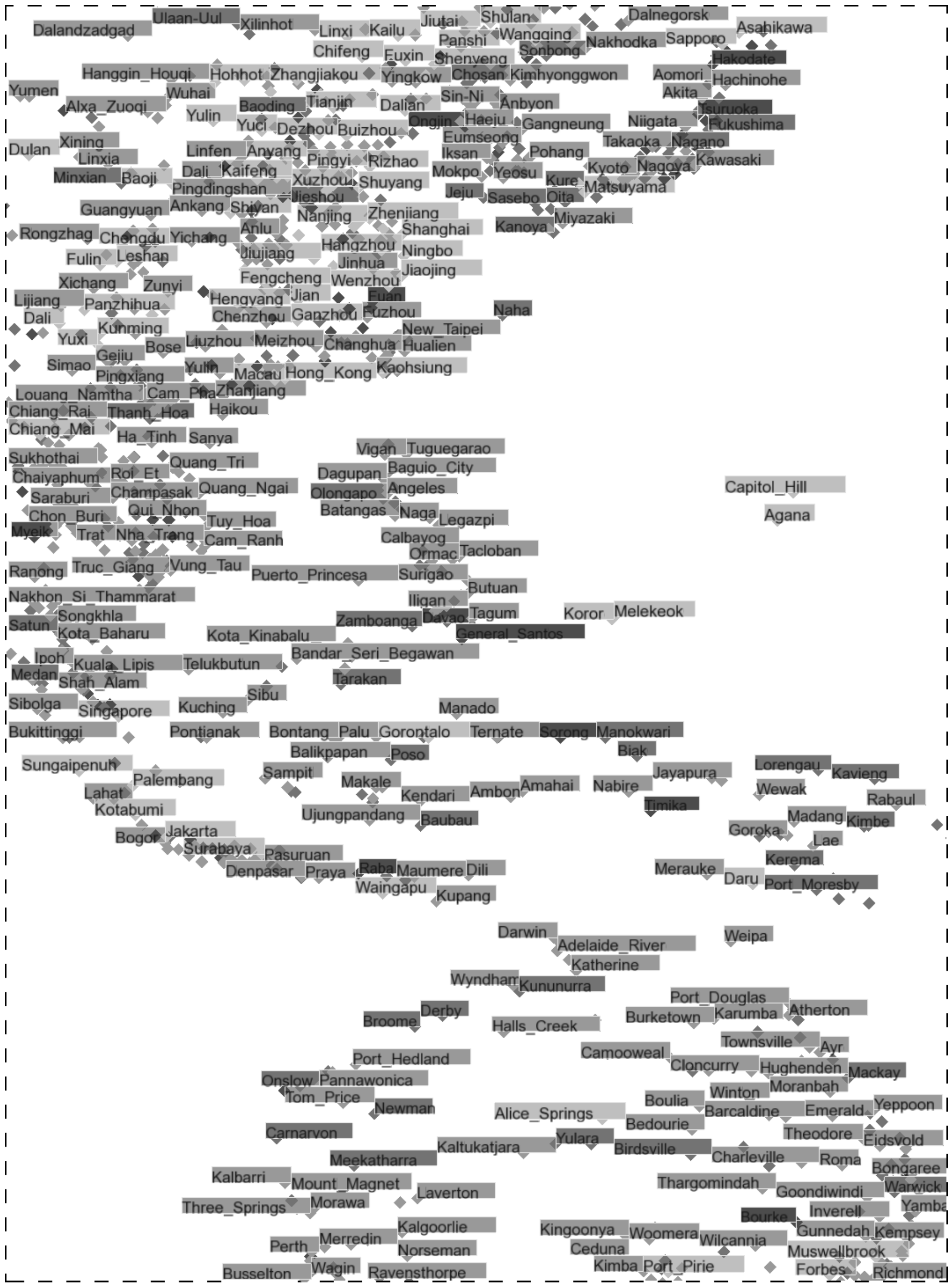


Figure 4.21: A map of isles between China and Australia labeled with our algorithm.

Part II

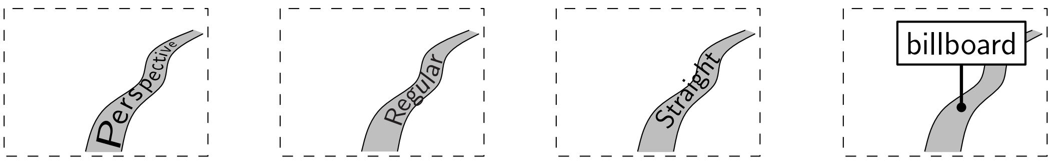
Labeling Line Features in Interactive Maps

Chapter 5

Labeling Streets with Embedded Labels

In large-scale paper maps, streets are commonly labeled embedded, that is, the label is placed inside the area occupied by its street and follows the curvature of the street. In interactive maps that allow for a 3D view, embedded labels are either rendered perspectively (see Figure 5.1(a)), or they are rendered regularly, that is, without perspective distortion (see Figure 5.1(b)). The advantage of embedded labels is that they reflect the course of a road and that they cannot occlude any other map object. The disadvantage is that an embedded label is hard to read if it is placed at a curvy part of its corresponding street. Besides, placing character by character has a high computation time. Another possibility for labeling streets is to place a label at a straight line that has a similar rotation as its corresponding street; see Figure 5.1(c)). The advantage of a straight label is that the computation time for such a labeling is rather low and it improves the legibility of the label text. The disadvantage is that the label–object association might get lost and that the course of the street is occluded. This is unfavorable for maps that are used while driving a car. The last type of street labels, we want to point out, is a billboard; see Figure 5.1(d). Billboards are used in some built-in car navigation systems. The advantage is that horizontally-written text is well legible [Tin72, KN85, WB05] (see the related-work section of Chapter 2). Moreover, in the case of a semi-transparent label background (which does hardly affect the legibility of the label [HV96]), billboards do not hide the course of the streets. Compared to embedded labels, the label–object association is worse, though.

Until today, there is only little work on street-label placement for both the static and the dynamic case. We intend to reduce this gap. To this end, in this chapter, we consider



(a) embedded, perspective (b) embedded, regular (c) straight, regular (d) billboard

Figure 5.1: Different ways to label streets in a large-scale map in a perspective view.

the problem of attaching embedded labels to streets with spatial extent. The challenge is to determine nice-looking label positions and, at the same time, to avoid overlaps of labels at junctions. In order to improve the information content, we split streets such that our complete street network only consists of polygonal lines (or, polylines for short). Even though we want to label many polylines, in our work about street labeling, we do not aim at placing more than one label per polyline. (We motivate this decision in the related work section below.) Therefore, we label as many *different* streets as possible. The corresponding static line-labeling problem StatLineLab is similar to the point-labeling problem StatPointLab of the previous chapter. Recall that we aimed for maximizing the sum of the weights of placed labels whereas we required that no two labels overlap. Instead of a weight for each point feature (where each candidate has the same weight), for StatLineLab, we consider the *cost* for each candidate of each (unweighted) line feature. The cost of a label candidate ℓ is meant to be small if ℓ is aesthetically pleasing, for example, if ℓ has few bends. For StatLineLab, we consider two goals: primarily, we want to maximize the number of placed labels whereas we require that at most one label per polyline is placed and that the labeling is overlap-free at junctions; secondarily, we want to select a cheapest candidate for each polyline, that is, we want to minimize the sum of all costs. Note that the primary goal is more important, that is, we prefer more probably bad-legible labels to fewer good-legible labels. We must split our goal as only maximizing the number of placed labels might lead to an unaesthetic labeling; on the other hand, we obtain total costs of zero if we place no label. We can formulate StatLineLab as follows:

> Given a set P of polylines in the plane and, for each $\pi \in P$, a set $L(\pi)$ of candidates and, for each $\ell \in L(\pi)$, its cost $c(\ell)$, find a subset $\mathcal{L} \subseteq \bigcup_{\pi \in P} L(\pi)$ of label positions that primarily maximizes the number $|\mathcal{L}|$ of placed labels and secondarily minimizes the sum of the costs $C = \sum_{\ell \in \mathcal{L}} c(\ell)$ such that no two labels intersect and $|L(\pi) \cap \mathcal{L}| \leq 1$ for each $\pi \in P$, that is, there is at most one label per polyline.

Rather than using static paper maps, most people nowadays use digital maps. We aim for placing street labels in a dynamic setting where the user can manipulate the view by continuously panning, zooming, rotating, and tilting. Every interaction changes the visible part of the map and thus the current labeling should be adapted suitable. Also when labeling streets, the additional challenge of a dynamic setting is to react to interactions appropriately and in real time. We stress that, in our model, we only know the interaction that is currently executed by the user.

Our Model. We continue the definition of the rectangular region R_i that is visible for the user at any given time t_i (see Chapter 4). Our scenario in this chapter additionally allows for rotation operations, that is, when rotating, R_i is rotated. Moreover, we allow for a 3D view. We observe that the rectangular region transforms to a trapezoid when tilting the map. More precisely, if we change the currently visible part of the map such that we can see more of the map at the horizon, the edge of R_i that corresponds to the bottom edge of the view gets shorter, the other base edge gets longer, the two angles at the shorter base edge of R_i gets larger, and the legs get longer; and vice versa.

Due to the change of the view, on a street, a label candidate that yields lower costs than the selected one can appear. Still, we prohibit that a label slides along its street to reach a better position. We believe that such moving labels overly attract the attention of the user. This can be dangerous, for example, when driving a car. The following extension of the static line-labeling problem brings us to the final definition of our line-labeling problem DYNALINELAB.

> For $i = 1, \dots, h$, let $\mathcal{L}_i$ be the subset of labels (at most one label per street) that are placed at time t_i and intersect R_i, and let $\mathcal{C}_i = \sum_{\ell \in \mathcal{L}_i} c(\ell)$ be the sum of the costs of the selected label positions in $\mathcal{L}_i$. Our primary goal now is to maximize the number of placed labels $\sum_{i=1}^{h} |\mathcal{L}_i|$ over all frames and our secondary goal is to minimize the sum of the costs $\sum_{i=1}^{h} \mathcal{C}_i$ over all frames.

Note that we obtain STATLINELAB out of DYNALINELAB if we restrict the setting to one point in time and choose a view that is large enough.

Even the problem of maximizing the number of placed labels in *one* frame is NP-hard. Consider the special case of the labeling problem where every label has exactly the same length as its street. That is, there is only one possible position for each label. Then, every overlap-free labeling corresponds to an independent set in the line intersection graph that contains a node for each street and an edge for each two intersecting streets. Since finding a maximum independent set in a line intersection graph is NP-hard [KN90] (even if every line is a straight-line segment and there are only three different line slopes), finding an overlap-free labeling with the maximum number of labels is NP-hard, too. Furthermore, if the labels can be shorter than the corresponding streets and each label is allowed to slide within its corresponding street, labeling a maximum number of streets without label–label overlaps has been shown to be NP-hard, even if every street is either horizontal or vertical [SU00]. For these reasons, we have developed a heuristic that computes nice-looking labelings (at least in our opinion). We guarantee that our labelings are overlap-free if the font height is bounded by the street width; in other words, we have to specify a suitable font size and constrain the minimum map scale. Otherwise, labels of streets that lie close together might overlap. To prevent such overlaps, a time-consuming collision detection would be needed.

Our approach differs from common digital map services in some points. First, we consider it somewhat annoying if the label of the same street is repeated several times whereas sometimes the distance between two such labels is quite small. Instead of repeating the label, we prefer that a label that left the view is placed inside the view again. Moreover, many online map services do not consider the history of the labeling; instead of updating the labeling locally, from time to time, the labeling is computed from scratch. This results in jumping labels that might disturb the users. Several digital map services use tile-based maps that precompute the labeling for several tiles. Sometimes, when a label overlaps two tiles, the pieces of the label do not fit. Moreover, caching makes it harder to react to user interactions, for example, when a label leaves the view.

Our Contribution. We present the first online algorithm that annotates streets with embedded curved labels (see Section 5.2). Our algorithm deals with panning, zooming, rotation, and tilting operations. We guarantee that our labelings are overlap-free if the font height is bounded by the street width. Our labelings are aesthetic in that we punish label positions with strong bends. We also present our algorithms for visualizing the map and rendering curved text (see Section 5.3). We have implemented our heuristic for solving DYNALINELAB and tested it on real-world data (see Section 5.4). Finally, we sketch ideas for further improving our algorithm; in terms of speed or aesthetics of its output, for example, by supporting the placement of more than one label per street or by studying the weighted case (see Section 5.5). We have made a video that shows our labeling algorithm in action[5.1].

5.1 Previous and Related Work

As stated in the introduction of this thesis, there are three types of map objects that are typically labeled: point, line, and area features. Wagner et al. [WWKS01] and others suggest solving the general label-placement problem by first generating a sufficient number of candidates for any map object to be labeled and then having a high-level algorithm making a choice between these positions such that as many map objects as possible receive an overlap-free label. In this chapter, we focus on the first step and use an ad-hoc method for the second step: we proceed incrementally, that is, we go through the map objects (streets in our case) in an arbitrary order and greedily place labels.

Edmondson et al. [ECMS97] introduce an algorithm for annotating rivers by rectangular labels. They first compute candidates by discretizing the polyline to be labeled. As the label is straight, it spans part of the river. The authors evaluate candidates by comparing the course of the spanned part of the river with the baseline of the label. The straighter the river is locally, the better the label position.

In his seminal work, the Swiss cartographer Imhof [Imh75] also lists rules for good line-feature labeling. He states that labels should follow the curvature of the lines features but avoid too strong bends. He also recommends to not put too much white space between consecutive characters of the same label. Our labeling algorithm observes Imhof's rules mentioned so far. Moreover, labels should be written as horizontal as possible. Due to the dynamic setting where the user can rotate the view, the orientation of objects can change quite often. Therefore we ignore this rules. Finally, a label should be repeated, especially if two objects are connected and cannot be distinguished. We also disregard this rule due to the fact that it is NP-hard to maximize, in an overlap-free *static* labeling, the number of *connecting arcs* that are labeled [GNN14]. A connecting arc is the part of a polyline that links two junctions.

Other than Edmondson et al. [ECMS97], Wolff et al. [WKvK$^+$00] respect Imhof's rule that a line label must follow the course of the polyline to be labeled. The authors label rivers externally. To this end, they compute stripes in which a label finally will be placed.

[5.1] `http://lamut.informatik.uni-wuerzburg.de/dynalinelab.html`

First they connect the start and the end point of the line with a circular arc. Then they iteratively transform the arc into a sequence of arcs that become closer and closer to the river. Additionally, the authors propose three measures to find aesthetic label positions within a stripe: the distance between the label and the river, the curvature of the label, and the number of inflection points of the label.

Strijk [Str01] presents one of the few algorithms for (static) street labeling with embedded labels. His work has inspired our algorithm. In order to obtain an overlap-free and aesthetic labeling, Strijk first computes candidates for each street. Next, he applies a heuristic for optimizing the evaluation function for the *entire* labeling (we, by contrast, decide street-wise). His function considers three criteria: association between the street and the label, label visibility, and aesthetics. Strijk also takes into account ways to split street names into parts. This enables him to sometimes leave junctions free. If a label of a short street consists of at most three parts, Strijk places one of the parts above the street, one into the street, and the last one below the street. For knowing where the algorithm can split a street name, Strijk creates a data base holding this information. We do not split street names as we aim for an almost universally usable algorithm (some exceptions exist; for example, the Chinese script should always be written straight-line). Note that, in an interactive scenario, users can zoom in if a street lacks a label.

To the best of our knowledge, only two approaches for labeling streets dynamically have been suggested so far; one by Maass and Döllner [MD07] and one by Vaaraniemi et al. [VTW12]. Both, however, place only straight labels.

Maass and Döllner [MD07] label interactive 3D virtual environments. They prevent label–label and object–label occlusions. For resolving label–label overlaps, they use a conflict graph. For object–label occlusions, they determine candidates and evaluate them by means of a visibility function. Thus, the main idea of this algorithm is similar to the one of Strijk. Their algorithm focuses on Manhattan-type city maps, that is, they do not treat curved streets. The authors support two modes of user interaction. In the first mode, the labeling does not change while the user manipulates the view. As soon as the user stops interacting, labels slide through the streets until they are visible (again). In the second mode labels fade out when the user begins to manipulate the map and fade in when the user stops the interaction. In our opinion, both options are not useful for navigation systems where the content of the view changes continuously. But also in other map applications it is disturbing if the user has to retrieve a certain label.

Vaaraniemi et al. [VTW12] give a *force-directed* algorithm for placing straight labels onto curved streets in real time. Repelling forces push overlapping labels away from each other; attracting forces pull labels to their reference points.

Note that the problem of labeling area features sometimes is solved by means of labeling line features. Area labels usually have extremely smooth bends. This is due to the fact that in the input data of an area object there is no prescribed line for the label. It must be determined first; for example, with the help of the medial axis [Kre94]. By contrast, in our algorithm, curves are directly given by the input data and thus can be quite irregular.

5.2 Labeling Algorithm

In short, our algorithm repeatedly tests if it can attach a label to an unlabeled street in the view. If we find *several* candidates, we try to select an aesthetic one (recall that we only aim for *one* label per street). Regardless of whether a polyline is labeled or not, the running time of the labeling test is linear in the number of the segments of the polyline (see Section 5.2.4). In the remainder of this section, we describe how to find aesthetic label positions and how to maintain a good labeling if the user interacts.

5.2.1 Finding Nice-Looking Label Positions

Any long-enough street contains an unbounded number of label candidates. For finding a good position, we initially place a label at the start point of its street. Then, we push the label through the entire street; see Figure 5.2. Simultaneously, we evaluate each label position by some *evaluation criteria*. By applying such a criterion, we obtain a cost, that indicates the quality of a position. Based on the costs, we select a good position. For the moment, we consider the computations of costs as black boxes.

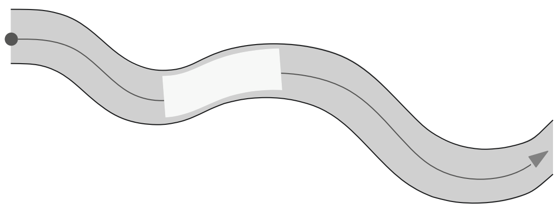


Figure 5.2: Pushing a label through its street.

We assume that a polyline $\pi = \langle s_1, \dots, s_m \rangle$ is given by an ordered sequence of segments, where the end point of segment s_i, $i = 1, \dots, m-1$, and the start point of segment s_{i+1} are the same. We call this point b_i a *bend*. Start and end points of streets are not considered bends. Let us anticipate that, in the implemented variant of our algorithm (that we present in this section), costs can only change at bends.

Evaluation. We first define the cost $C(b_i)$ for each bend b_i. The cost $C(b_i)$ depends on several criteria, which are weighted by importance. We sum up the weighted costs of every criterion in order to obtain the cost of a bend; formally,

$$C(b_i) = \sum_{e \in E} w_e c_e(b_i)$$

where e is an evaluation criterion from the set of criteria E, w_e is the weight of e, and $c_e(b_i)$ is the cost of e for bend b_i that lies between the segments s_i and s_{i+1}.

Next, we compute the cost of a label position. Consider a polyline $\pi = \langle s_1, \dots, s_m \rangle$ and any label position ℓ at π. Let $\pi' = \langle s_j, \dots, s_k \rangle$, $1 \leq j \leq k \leq m$, be the sequence of segments that ℓ occupies. As costs only can change at bends, we simply sum up the combined costs for each bend in π' in order to evaluate ℓ. Formally we compute

$$\sum_{i=j}^{k-1} C(b_i).$$

Suppose that ℓ does not start at bend b_{i-1}, the start of s_i, but anywhere else at s_i. Then ℓ has at least the same cost as a label of the same length that does start at b_{i-1}; see Figure 5.3. For that reason, we can discretize the polyline while still evaluating each possible position as it suffices to consider the bends as starting points for label position.

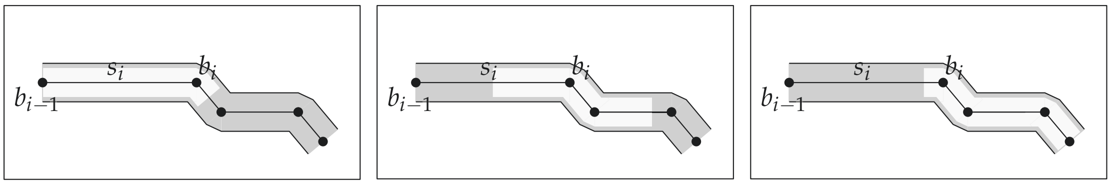


Figure 5.3: A label that starts anywhere between b_{i-1} and b_i can only have a higher cost than a label that starts at b_{i-1}.

Now, for each bend $b_i, i = 1, 2, \ldots$, we search for an index $k(i) \geq i$ (if exists) by collecting the consecutive segments $\langle s_i, \ldots, s_{k(i)} \rangle$ such that the sum of all lengths of the collected segments is just larger than ℓ, or formally, such that

$$\sum_{j=i}^{k(i)-1} |s_j| < |\ell| \text{ and } \sum_{j=i}^{k(i)} |s_j| \geq |\ell|,$$

where $|\ell|$ is the length of ℓ and $|s_j|$ the length of s_j. For each sequence $\langle s_i, \ldots, s_{k(i)} \rangle$, we compute its cost (as described above) and determine the cheapest sequence. As the sequence usually is longer than the label, we center the label within its sequence. See Figure 5.3 again: it is $k(i) = i + 1$. As long as we slide a label within the sequence $\langle s_i, s_{i+1} \rangle$, the cost does not change.

By reusing an already-computed evaluation, we can determine the cheapest label position in $\mathcal{O}(m)$ time (recall that m is the length of the currently evaluated polyline). Consider a sequence that we have just evaluated. In order to obtain the next sequence, we remove the very first segment from the sequence. We adjust the cost accordingly. We then add further segments to make the sequence again longer than the label to be placed and adjust the cost. This way, we handle each bend of a polyline to be labeled only twice—one time, we add it to the sequence, one time, we remove it. It remains to ensure that any evaluation only needs constant time.

We now describe our evaluation criteria.

Junction Criterion. As we want to place as many labels as possible, we try to avoid labels that pass a junction. Otherwise, a placed label A might block a junction for another label B although there is a junction-free position for A but none for B. For that reason, we charge a constant cost of X for every junction (or crossing) that a label position contains. The more important it is to avoid junctions (compared to irregular courses), the higher the cost. If we find a junction while evaluating a sequence, we test whether it is already occupied. If so, we cancel the evaluation of the current sequence and start a new sequence after the junction.

Observe that we visualize our streets with spatial extent. By contrast, our graph data structure for the street network consists of points (junctions) and polylines (connecting arcs). For that reason, our evaluation takes the *visual* start and end points of the streets into account and not the point that is given in the data structure; see Figure 5.4. That is, two streets A and B that touch such that the end point of A is the same as the start point of B do not form a junction. If the start/end point of a street A touches a bend of another street B, like at a fork, these streets do not form a junction. Instead, for evaluation, we omit that very part of A that overlaps (the spatial extent of) B. If a bend of street A shares a point with a bend of street B, then A and B form a junction.

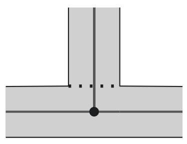


Figure 5.4: Data structure (lines) and visualization (bars) of our street network. The dotted line indicates the visual start or end point of the vertical street.

Angle Criterion. Let $\alpha_i \in (-180°, 180°)$ be the angle between the two consecutive segments s_i and s_{i+1}, measured as shown in Figure 5.5. In the sequel, we identify α_i with its absolute value $|\alpha_i|$. Obviously, the larger α_i, the heavier the bend of the street, the worse a label looks if it contains bend b_i.

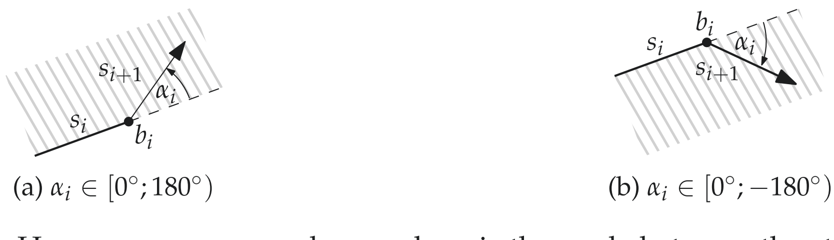


Figure 5.5: How we measure angles: angle α_i is the angle between the straight line induced by s_i and s_{i+1}.

This observation immediately leads to the idea to use a value proportional to α_i as part of the cost of a bend b_i. We further observe that a curvature (and such a label that contains the curvature) looks smoother if it consists of many angles of the same sign that sum up to an angle α compared to a single bend with an angle α. In order to punish one large angle more than several small angles, we use costs of α_i^2; see Figure 5.6. We

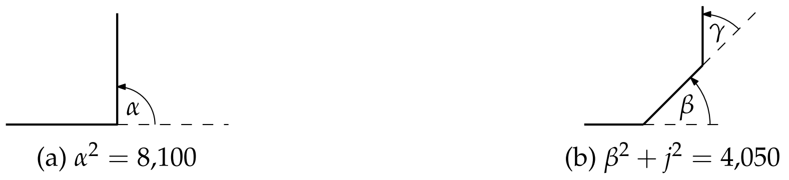


Figure 5.6: We use the squared angle in order to punish large angles more than several small angles.

moreover punish very short segments by means of the angle. Consider three consecutive segments. Let $\alpha = 45°$ be the angle of both bends. It is easy to see that the smaller the intermediate segment, the worse looks a label that traverses the three segments. For our evaluation, we solve this problem by collecting too short consecutive segments and squaring the sum of their angles. With the square of the sum rather than the sum of the squares, we increase the penalty. We visualize this approach in Figure 5.7 and formalize it in the following. Let τ be a threshold that defines the desired minimum length of a segment (for instance, let τ be the average width of a character of the given font). For determining the angle $\alpha_i,\ i = 1, \dots, m-1$, between the segments s_i and s_{i+1} that contributes to our evaluation, we first take the length $|s_i|$ of s_i into account. More precisely, we search for an index $k(i) \geq i$ (if exists) such that the sum of the lengths of the segments of the sequence $\langle s_i, \dots, s_{k(i)} \rangle$ is just larger than τ:

$$\sum_{j=i}^{k(i)-1} |s_j| < \tau \text{ and } \sum_{j=i}^{k(i)} |s_j| \geq \tau.$$

We define $\alpha_m = 0$. The final cost for the subsequence $\langle s_i, \dots, s_{k(i)} \rangle$ is

$$\left(\sum_{j=i}^{k(i)} \alpha_j \right)^2 + x \cdot X$$

where x is the number of the corresponding junctions. Note that it suffices to consider

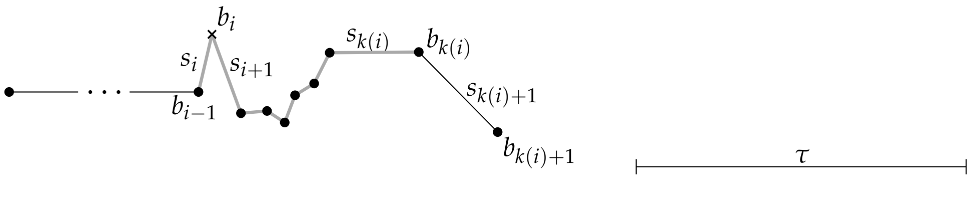


Figure 5.7: Assume that b_i is the next bend to be evaluated. We search for an index $k(i)$ such that the sequence $\langle s_i, \dots, s_{k(i)} \rangle$ is just longer than τ (note that we sum up the lengths instead of using the euclidean distance). For the total cost, we include all bends that are contained in the sequence but b_{i-1} as it was already considered in the previous evaluation.

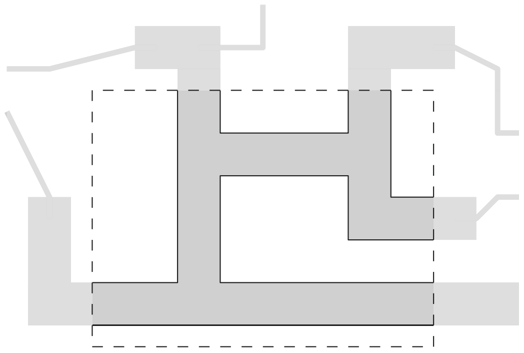
succeeding segments; we evaluate sequences containing preceding segments by starting evaluations at bends that precede the currently evaluated bend. We remark that we do the combination of too short segments on runtime rather than smoothing the input data in a preprocessing as, on the one hand, the threshold τ depends on the current scale; on the other hand, if we smooth the input data, we omit some potential label positions.

Even when recycling evaluations, this method has a running time of $\mathcal{O}(m^2)$ for finding a nice-looking label position in one polyline. Consider the case that each segment has the same length and two segments are needed to satisfy the threshold τ. Assume that the length of the corresponding label ℓ is half the length of the polyline, that is, ℓ overlaps $m/2$ segments; consequently, ℓ overlaps $m/4$ subsequences. Whenever, we start an evaluation at the next bend, we have to update each sequence. The claimed running time follows.

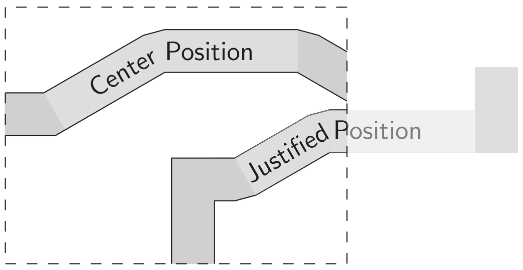
In order to obtain a running time of $\mathcal{O}(m)$, we introduce two possibilities. In the case that, due to the evaluation of the next bend, the first subsequence becomes shorter than τ, we could (i) just ignore that the subsequence is too short or (ii) merge the first and the second subsequence. For both cases, before starting an evaluation at each bend, we iterate the street in order to divide the polyline into subsequences and to compute the square over the sum of the angles for each subsequence. While evaluating the candidates, for case (ii), we have to extract the root from the squared sum of the first and probably of the second subsequence and correct the values. The running time, however, is satisfied. We cannot predict which of the two versions yields nicer results. In case (i), we usually underestimate the cost; in case (ii), we usually overestimate the cost.

At last, we define a maximum angle $\alpha^\star$. Whenever the angle of two consecutive segments in a sequence is larger than $\alpha^\star$, we stop evaluating the current sequence and start with the next one.

View Criterion. So far, our algorithm labels a complete street network. This is certainly unfavorable for dynamic scenarios as it is very likely that labels of streets within the view lie outside the view. For that reason, in every frame, we collect for each polyline all its segments that intersect the view. If a polyline leaves and enters the view multiple times, we only collect the longest visible and connected part. It also would be conceivable to evaluate all visible parts and choose the cheapest label position. Further we permit labels to overlap the view boundary. On that account, for each end of a polyline that leaves the view, we expand the collection of visible segments by additionally collecting the adjacent segments of the polyline that lie outside the view until the sum of the lengths of the new collected segments is just larger than some threshold v (or less if the remaining part of the polyline that lies outside the view is too short); see Figure 5.8(a). In order to avoid labels that lie completely outside the view, we recommend to choose v dependent on length of the corresponding label. The idea is that sometimes the course of a polyline might be quite nasty within the view but smooth at the view boundary. As users can interact with the map, they can pan such that the label becomes completely visible if desired. We evaluate the (extended) visible parts of a polyline but we punish sequences overlapping the view boundary with some additional cost V. If the finally selected sequence overlaps the view boundary, we do not center the label within its sequence but we place the label at that very end of the sequence that lies within the view (such that the label lies within its sequence); see Figure 5.8(b). Hence, we maximize the visibility of the label.



(a) the thick lines show the (extended) visible parts of the polylines; the thin lines indicate somewhat more of the street network



(b) positioning a label within its sequence

Figure 5.8: If a polyline leaves the view, we expand its collection of visible segments. We maximize the visibility of a label by positioning it as much within the view as possible.

5.2.2 Dealing with Interactions

Providing interactive maps, we have to deal with interactions properly. Independently of the precise interaction, *in each frame*, we try to label every unlabeled street by means of our evaluation routine. There are mainly three reasons for unlabeled streets: First, due to interactions, new parts of the map become visible. Conversely, labels might leave the view. We require that, for a better user orientation, a label that has left the view completely appears at another position within the view. The last reason is that we could not find any permitted label position at the street so far.

Panning. Panning is the most general case. New parts of the map become visible while other parts leave the view; see Figure 5.9. We just apply our labeling algorithm to every unlabeled street.

Zooming Out. If users zoom out, they can see a larger part of the map; that is, the map scale decreases. As a consequence, streets get smaller. We require, however, that the size of a label remains constant on the screen while users zoom. Recall that we place a label in a sequence of segments. As streets get smaller, also the lengths of the sequences get shorter. We either update a sequence that becomes too short to host its label or we try to place the label at another position. Thus, we can label some unlabeled streets as several junctions lose their labels, that is, junctions are free for the labels of the

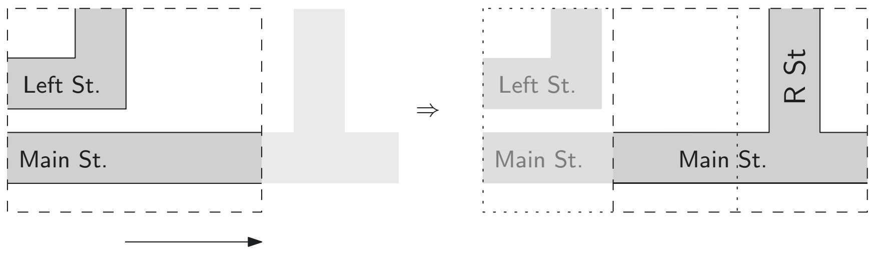


Figure 5.9: If we pan to the right, on the left, labels vanish; on the right, streets appear. We try to label streets that are unlabeled within the view.

crossing streets. Similar to a panning operation, new (so far unlabeled) parts of the map become visible. In the following, we give a detailed approach how to expand sequences and how to find a suitable position for the label within the grown sequence.

If a sequence becomes too short to host its label, we repeatedly append a segment at the beginning or at the end of the sequence until the sequence is again large enough to host the label. In which direction we expand the sequence depends on a) involved junctions and b) the segment's location with regard to the view. For the sake of simplicity, we refer to the segment that precedes the sequence as *left* segment s_l and to the segment that succeeds the sequence as *right* segment s_r. Let $t(s_l)$ and $t(s_r)$ be the touching points of s_l and s_r with the sequence; see Figure 5.10. First assume that both $t(s_l)$ and $t(s_r)$

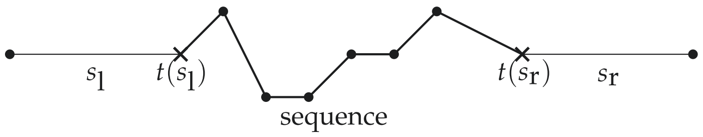


Figure 5.10: The segment that precedes the sequence is s_l; it touches the sequence at $t(s_l)$; analogously for s_r.

touch a junction. If both junctions are free, we enhance the sequence into the direction of the end of the label text (that is, into the reading direction) in order to keep one junction free; see Figure 5.11(a). If the junction at $t(s_r)$ is already occupied by a label, but $t(s_l)$ is free, we expand the sequence to the left and vice versa (this case is similar to Figure 5.11(a)). If both $t(s_l)$ and $t(s_r)$ touch an occupied junction, we remove the label and apply our evaluation routine in order to try to find a new position for the label; see Figure 5.11(b). Now assume that neither $t(s_l)$ nor $t(s_r)$ touches a junction. We prefer to append a segment that lies completely within the view to a segment that intersects the view boundary to a segment that lies completely outside the view (in order to maximize the visibility of the label). If s_l and s_r have the same property with regard to the view, we first select the longer segment, say s_l. Next, we add segments at the right until the sequence of the added segments is longer than $|s_l|$. Then we again expand the sequence to the left, and so on. We grow the sequence in such an alternating manner until the sequence is finally long enough. If we find a segment that touches a junction or intersects the view boundary during this process, we act as described before.

Note that we grow sequences as long as possible rather than computing a new label position in order to avoid jumping labels as they might overly attract the user's attention. Consequently, we accept possibly bad-looking label positions. We reject a label position if it contains at least one bend that exceeds the maximum angle; then, we apply our evaluation algorithm for the corresponding street.

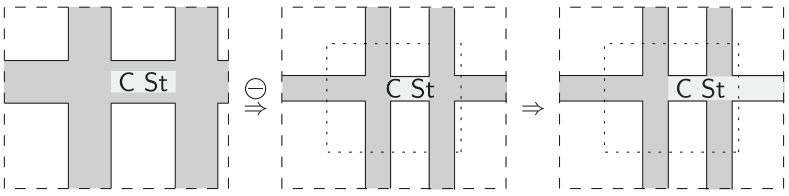
It remains to select a proper label position within the possibly unequally expanded sequence. Due to the shrinking of the streets while users zoom out and maintaining a constant label size on the screen, it seems as if a label would *grow* within its street. Consider the case that the expanded sequence does not intersect the view boundary. As before, we aim for a central position of the label within its sequence. The reason is that in most cases the expanded sequence is longer than the label such that the label can grow equally to both of its sides. We consider an equally growing label to be more aesthetic.

Assume that a label is centered within its sequence. Now the sequence is expanded equally to both sides. Then the new position of the label is also at the center of the expanded sequence inducing an equally growing of the label; see Figure 5.12(a). If, however, the sequence was expanded unequally, just centering the label within the expanded sequence might lead to jumps, for instance, if we added only one segment that is longer than the label; see Figure 5.12(b). On that account, we set the bottom corner of the currently placed label that is *further* away from the center of the expanded sequence (measured along the sequence) as *fix point*. We now try to shift the label at the center of the expanded sequence. If the centered label overlaps the fix point, we found the new label position. This equals an unequally growing over a short time resulting in a centered position without causing jumps. If otherwise the label does not overlap the fix point, we shift the label such that the same label corner as in the old label position touches the fix point; see Figure 5.12(c). The idea is that, regarding such a positioning over several frames, it seems as if the label first grows into the direction of the center of the sequence. After reaching the center with one corner, the label grows further one-sided until the label is centered. Then, it grows equally.

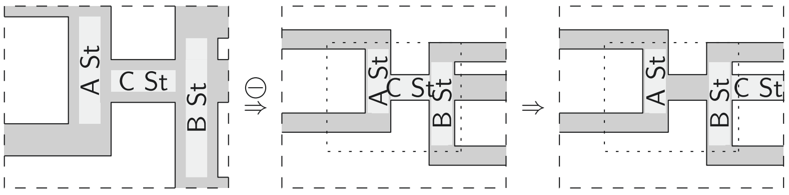
Now consider the case that the sequence intersects the view boundary; we aim for maximizing the visibility of the label. Without loss of generality, let the sequence intersect the right boundary. Then the fix point is at the right bottom corner of the placed label. We shift the label with its left end to the left end of the updated sequence. If the label overlaps the fix point, we found the new label position. Otherwise we shift the label such that the right corner of the label touches the fix point. With this, the label smoothly grows into the direction of the view.

There is certainly a large number of different approaches for expanding the sequence and placing the label. Our approach sometimes leads to an imbalanced grow. Nevertheless, we do not force an overall equally growing as this would only waste computation time without a real visual effect.

Zooming In. We observe that, while zooming in, the map scale increases, that is, streets get larger; reverse Figure 5.11. Therefore, we can label some unlabeled streets

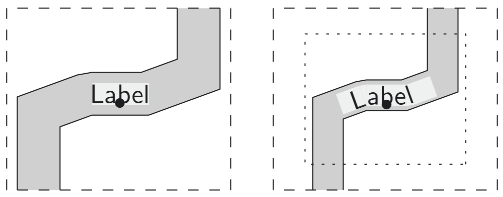


(a) as both adjacent junctions are *free*, the sequence grows into reading direction (*here*: to the right); similarly, if the junction on left is occupied, we expand the sequence to the right, too

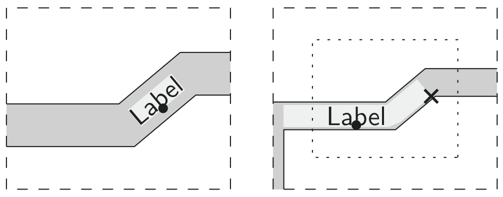


(b) as both adjacent junctions are *occupied*, we compute a new position for "C St"

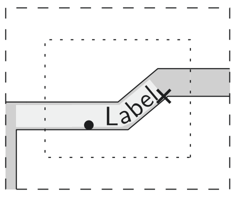
Figure 5.11: From left to right: if users zoom out, the map scale decreases. We have to grow each sequence that becomes shorter than its label; here, it is "C St". The lighter parts show the sequences, the dotted rectangle indicates the former view.



(a) in the case of a centered label and an *equally* extended sequence, the label position stays centered



(b) in the case of a centered label and an *unequally* extended sequence, centering the label might cause jumps



(c) to avoid jumps, place the label at its fix point inducing a one-sided label growing

Figure 5.12: A proper positioning of the label within its sequence results in suitable growing labels while zooming out. In (a) and (b), the left figures show the initial situation. In (c), we resolve the conflict from (b). The lighter parts are sequences, the dotted rectangle indicates the former view, the points show the center of the sequence, and the crosses show the fix points.

that now offer enough space to host their labels. On the other hand, some streets get unlabeled as their corresponding labels leave the view completely.

Our approach for zooming in is similar to the inversion of the case that the user zooms out. As before, we require a constant label size on the screen. Consequently, the sequences of segments might become much longer than the corresponding labels. This is unfavorable as we base many of our decisions on the sequences in order to save computation time. Growing streets cause the visual effect that labels *shrink* within their sequences.

We first correct the label positions within the sequences, then we remove each segment that is no longer needed to host the corresponding label: assume that the sequence does not overlap the view boundary. If the label is already centered within the sequence, the label stays centered. Otherwise, we set the bottom corner of the label that is *nearer* to the center of the sequence as fix point. Regarding this replacement over several frames, in most cases, it seems as if a label first shrinks such that it becomes centered again and then it shrinks equally. (In rare cases, the label will never overlap the center of the sequence and thus it will never shrink equally unless the user zooms out again or the label leaves the view.)

If the sequence overlaps the view boundary, without loss of generality, say the right boundary, we shift the label such that the left end of the label touches the left end of sequence. With this, we keep segments outside the view free of their labels; thus, these segments are removed from the sequence.

Rotation. Due to a rotation operation, new parts of the map containing so-far unlabeled streets can get visible. Moreover, some labels might leave the view completely.

One of our aims is an easily legible labeling. For this, we have to care about the orientations of the labels. If the user rotates the view, the orientation of a label can become upside down. Unfortunately, there are cases in which the orientation of the label is not unique: correcting one part of the label results in writing another part of the label upside down; see Figure 5.13. As a rough guide, we determine the orientation by the gradient of the line between the start point and the end point of the label.

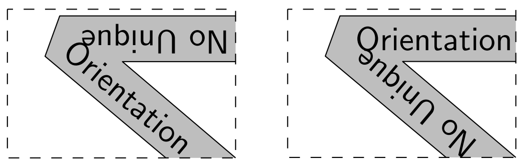


Figure 5.13: Correcting the orientation of one part of a label might lead to a distortion of another part. Based on the gradient of the line connecting the two ends of the label, our algorithm decides for the solution on the right.

5.2.3 Maps in a 3D View

In Chapter 2 we stated that Maass et al. [MJD07b] propose two ways for rendering labels in a 3D virtual environment; with perspective distortion and regular. Indeed, our algorithm can deal with both possibilities. It is a question of the input. We obtain labels with perspective distortion if we compute and place labels in world space. In that case, we can use our 2D algorithms also for labeling a map in a perspective view without any changes. If we compute and place labels in screen space, we receive a labeling with regularly-rendered labels. Then, we have to spent more computation time as each interaction (but panning horizontally) leads to growing streets in the foreground and shrinking streets in the background. We anticipate that we only implemented the rendering with perspective distortion.

5.2.4 Wrap-Up: Pseudocode and Running Time

Knowing all the details of our algorithm, we next present some pseudocode that summarizes all the steps of our algorithm. In the end, we analyze its asymptotic running time. Let P be a set of polylines, that is, P is the entire street network. We call Algorithm 5.1 in each frame. For the sake of readability, in Procedure 5.2, we omit the exit condition that, for each $\pi = \langle s_1, \ldots, s_m \rangle$, there is a segment s_i such that each sequence $\langle s_j, \ldots, s_m \rangle$, $j \geq i$, is too short to host the label of π. Equally, we leave out the recycling of already computed evaluations and subsequences. Recall that α_i is the angle between the segments s_i and s_{i+1}, that is, it is the angle at bend b_i. Furthermore, $\alpha^\star$ is the maximum angle, $|s_j|$ is the length of the segment s_j, V is the cost if a sequence overlaps the view boundary, and X is the cost if a sequence overlaps an unoccupied junction (there is no need to define a weigh for these two criteria; instead we can rise the cost). A proper label position depends on the interaction and the sequence's location with regard to the view.

Our algorithm has a total running time of $\mathcal{O}(|P| \cdot (M + C))$ where $|P|$ is the number of polylines of the entire street network, M is the number of segments of that polyline out of P that has the largest number of segments, and C is the number of characters of that label of a polyline in P that has the most characters. For each loop in Algorithm 5.1 (but that in line 9), it is easy to see that, no loop needs more than $\mathcal{O}(|P| \cdot (M + C))$ time. We already pointed out that our evaluation algorithm needs $\mathcal{O}(M)$ time for each polyline if we recycle evaluations of sequences and subsequences and refrain from recomputing subsequences in the case that the first subsequence becomes too short. As Procedure 5.2 shows, there is no other running time-critical part. Hence, the claimed running time follows immediately.

5.3 Visualization Algorithm

As OpenStreetMap[5.2] data is open-source, we decided to use OpenStreetMap maps as input for our algorithm. Nevertheless, the raw material is not suitable for our purposes.

[5.2] `http://www.openstreetmap.de/`, accessed Dec. 26, 2014

Algorithm 5.1: main()

```
// from: VIEW CRITERION
**foreach** *polyline* $\pi \in P$ **do** determine the visible part of $\pi$ and store it in the set of visible polylines $\mathcal{V}$    // $\mathcal{O}(|P| \cdot M)$
**foreach** $\pi \in \mathcal{V}$ **do** extend $\pi$ by segments outside the view if $\pi$ overlaps the view boundary    // $\mathcal{O}(|\mathcal{V}| \cdot M),\ |\mathcal{V}| \leq |P|$
store all labels that are still visible in the set of placed labels $\mathcal{L}$    // $\mathcal{O}(|\mathcal{V}|),\ |\mathcal{V}| \leq |P|$
// from: DEALING WITH INTERACTIONS
**foreach** $\ell \in \mathcal{L}$ **do**    // $\mathcal{O}(|\mathcal{L}|),\ |\mathcal{L}| \leq |P|$
    **if** *the user zooms out* **then** grow the sequence of $\ell$; remove $\ell$ from $\mathcal{L}$ if this is not possible    // $\mathcal{O}(M)$
    **if** *the user zooms in* **then** shrink the sequence of $\ell$    // $\mathcal{O}(M)$
    **if** *the user rotates* **then** correct the orientation of $\ell$ if necessary    // $\mathcal{O}(1)$
(9) **foreach** $\pi \in \mathcal{V}$ *that is not labeled* **do**    // $\mathcal{O}(|\mathcal{V}|),\ |\mathcal{V}| \leq |P|$
    $\mathcal{L} = \mathcal{L} \cup$ EvaluationProcedure($\pi$)    // $\mathcal{O}(M)$
// in: RENDERING CURVED LABELS (follows)
**foreach** $\ell \in \mathcal{L}$ **do** draw $\ell$    // $\mathcal{O}(|\mathcal{L}| \cdot C),\ |\mathcal{L}| \leq |P|$
$\mathcal{L} = \emptyset$    // $\mathcal{O}(1)$
```

Procedure 5.2: EvaluationProcedure(π)

Let *i_start* be the index of the bend at which we started the current evaluation and i the index of the currently considered bend. Then, C is the cost of the currently evaluated sequence $\pi' = \langle s_{i\_start}, \ldots, s_{i+1} \rangle$. Moreover, i' stores the first index when collecting too short segments, x counts the number of junctions, w_α is the weight for the angle criterion, and π^* stores the cheapest sequence.

```
$i\_start = 1$
$\pi^* = null$ // infinity cost
while we have not processed every segment of the polyline $\pi = \langle s_1, \ldots, s_m \rangle$ do
 REFRESH:
    $C = 0, x = 0$
    $i = i\_start - 1$ // to also consider sequences with one segment
    // NO-COST: one segment suffices
    if the sequence $\pi'$ is long enough to host its corresponding label then
        return proper label position within $\pi'$
    // ANGLE CRITERION
    while the sequence $\pi'$ is too short to host its corresponding label do
        $i = i + 1$
        if $\alpha_i > \alpha^*$ or $b_i$ is a junction that is already occupied then
            $i\_start = i + 1$ and go to REFRESH
        if $b_i$ is a junction then $x = x + 1$
        $i' = i$
        while $\sum_{j=i'}^{i} |s_j| < \tau$ and $\pi'$ is still too short to host its label do
            $i = i + 1$
            if $\alpha_i > \alpha^*$ or $b_i$ is a junction that is already occupied then
                $i\_start = i + 1$ and go to REFRESH
            if $b_i$ is a junction then $x = x + 1$
        $C = C + w_\alpha \cdot (\sum_{j=i'}^{i} \alpha_j)^2$
    // JUNCTION CRITERION
    $C = C + x \cdot X$
    // VIEW CRITERION
    if $\pi'$ overlaps the view boundary then
        $C = C + V$
    if $C$ is lower than the cost of sequence stored in $\pi^*$ then store $\pi'$ in $\pi^*$
    $i\_start = i\_start + 1$
return proper label position within $\pi^*$
```

We now describe how we preprocess the input data, how we visualize the street network, and we detail our rendering algorithm for curved text.

5.3.1 Preparing the Input Data

In the input data, streets are given as unordered lists of segments; each segment is given by two coordinates. As our algorithm is based on consecutive segments, we first order the segments such that each end point of a segment is the starting point of the next segment. Additionally, in reality, a street often is not a polyline but it forks. For that reason, we split such a street into two polylines and number the split streets serially by attaching a number to the street name; see Figure 5.14. We treat each split street as independent street. Furthermore, many streets are given several times—for instance, once as “residential”, once as “pedestrian”, and once as “cylceway”. In order to avoid clusters of streets, we exclude some categories for our final map. We also eliminate streets for which no name is given. By the removal of streets, we sometimes destroy the connection of the street network. We remove unconnected streets that consist of a single segment, but accept unconnected streets that consist of more than one segment.

As we use streets with spatial extent, we compute a second street network that we use for evaluations only. To this end, we copy the prepared network and cut segments at junctions (or rather forks) such that we obtain a street network that observes the visual start and end point of streets.

In order to save computation time, we compute the lengths of the segments and the angles between them in a preprocessing. At runtime, we query the values and adjust the lengths according to the current scale.

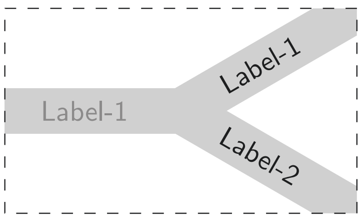
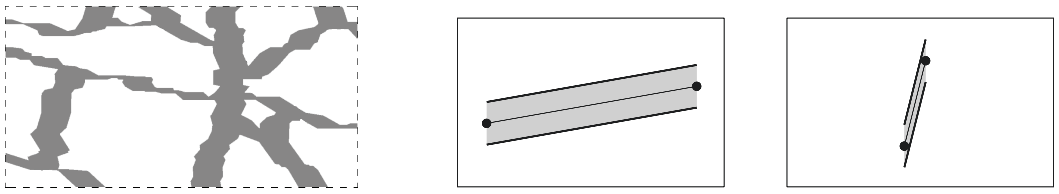


Figure 5.14: In reality, many streets fork.

5.3.2 Visualization of the Map

To find a quite nice visualization of the street network, we learned by trial and error. As, at least in our visualization frame work, there is no possibility to draw very thick lines in order to represent the street network, we have to draw rectangles instead.

We first tried to draw rectangles by translating the x-coordinate by just adding and subtracting some value ε, that is, for a segment $\langle (x, y), (x', y') \rangle$, we used $(x + \varepsilon, y)$ as top left corner, $(x - \varepsilon, y)$ as lower left corner, $(x' + \varepsilon, y')$ as top right corner, and $(x' - \varepsilon, y')$ as bottom right corner. We show the results in Figure 5.15.

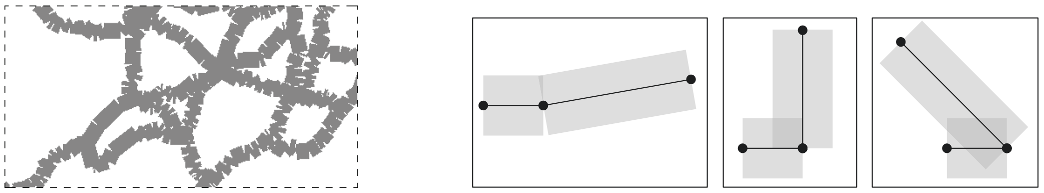


(a) street network (b) the rectangle strongly depends on the slope of the line connecting the two coordinates

Figure 5.15: Visualization of a street network by translating the x-coordinate.

Now it was clear that we draw axis-parallel rectangles and rotate them. This resulted in the next problem, namely gaps; see Figure 5.16. If we increase the length of a rectangle, some gaps are not closed, some rectangles cause jags as they are too long. Instead, we implemented two possibilities to close the gaps: at each bend, we placed (a) a triangle or (b) a circle (see Figure 5.17(a)). In our opinion, when closing gaps by circles, the visualization of the street network looks slightly smoother but it also needs much more computation time. When we closed the gaps with triangles, on a low-end computer, we reached a frame rate of 80 FPS. As in our visualization framework, there was no possibility to automatically draw circles, we approximated a circle by a 20-gon; see Figure 5.17(b). The frame rate dropped to 8 FPS.

Nevertheless, if we close each gap by a triangle, dead ends violate the appearance of the visualized street network. Moreover, when the ends of two streets touch, some gaps are not closed. On that account, we finally decided for a mixed solution: we close many gaps by triangles; we close the remaining gaps by placing a circle at each start and end point of a street; see Figure 5.17(c). This results in a frame rate of 56 FPS (which was sufficient on the low-end computer). We finally complement the streets by a border: we draw the network two times, whereas the network for the border is slightly larger than the network for the streets.



(a) street network (b) the larger the rotation, the larger the gap

Figure 5.16: Rotated rectangles cause gaps.

5.3.3 Rendering Curved Labels

As we are not aware of any bibliography for placing curved text that is suitable for our program, we implemented the text placement. For placing curved text, it is very

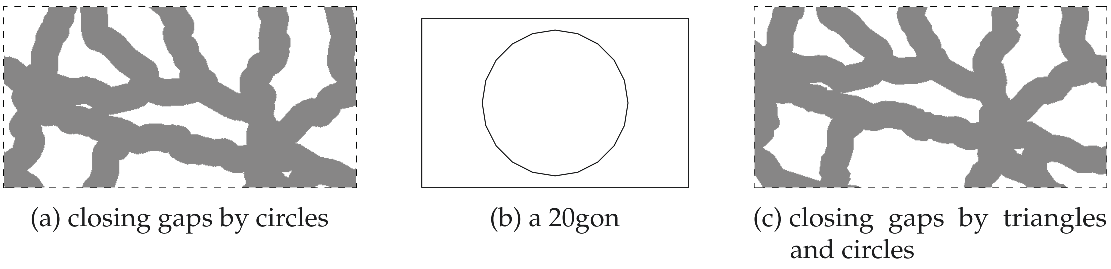

(a) closing gaps by circles (b) a 20gon (c) closing gaps by triangles and circles

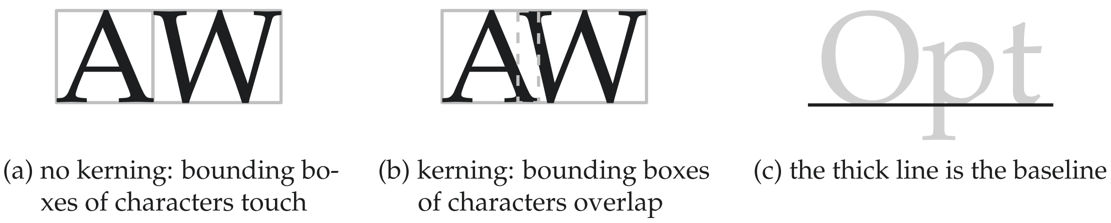
Figure 5.17: Closing gaps with circles and triangles.

intuitive to place the text character by character. This needs of lot of computation time, though. On that account, we place substrings. Consider a sequence that is long enough to host its label. Let $\mathcal{T}$ be the corresponding label text and let $|\mathcal{T}|$ be the *width* of the label text, that is, the distance between the left-most and the right-most point of the horizontally-written text. We first build a substring $\mathcal{S}$ out of $\mathcal{T}$ such that $\mathcal{S}$ is almost as long as the first segment s of the sequence; formally, we search for an index k (if exists) that defines the substring $\mathcal{S}$ such that

$$|S| = \sum_{j=1}^{k} |\mathcal{T}_j| \leq |s| \text{ and } \sum_{j=1}^{k+1} |\mathcal{T}_j| > |s|$$

where $|S|$ is the width of S, $|s|$ is the length of s, and $|\mathcal{T}_j|$ is the width of the j-th character of $\mathcal{T}$.

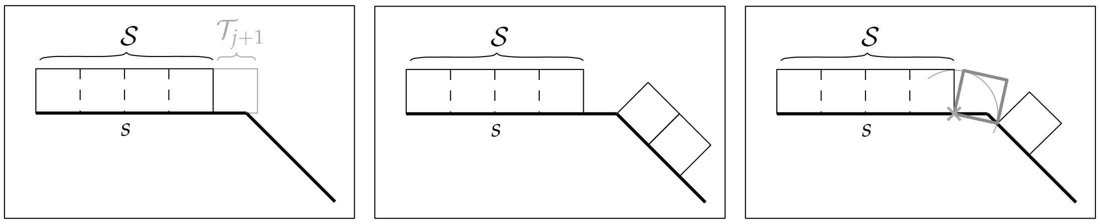
Placing substrings has another advantage: many visualization frame works support kerning, that is, the spacing between two consecutive characters is automatically adjusted as large spacings might irritate the reader; see Figures 5.18(a) and 5.18(b). Now we place $\mathcal{S}$ at the beginning of s such that the baseline of $\mathcal{S}$ (see Figures 5.18(c)) lies on s.



(a) no kerning: bounding boxes of characters touch (b) kerning: bounding boxes of characters overlap (c) the thick line is the baseline

Figure 5.18: Kerning and baseline.

As placing the next substring immediately at the beginning of the succeeding segment sometimes results in quite large spaces between the two substrings, we build an artificial segment: we place a circle with radius $|\mathcal{T}_{j+1}|$ at the right-most point of $\mathcal{S}$ that lies at s. With this, we know the start point an the end point of the artificial segment. We place $\mathcal{T}_{j+1}$; see Figure 5.19. We handle segments that are too short to host a single character equally. Also for the following characters, we repeatedly build a substring and an artificial segment (whereas we, of course, always exclude that part of the current



(a) place a sequence $\mathcal{S}$ which is almost as long as segment s (b) also consider the remaining part of s to avoid gaps (c) rotate T_{j+1} suitable

Figure 5.19: Rendering curved text.

segment that the character at the artificial segment occupies). Finally, we shift the label such that it is vertically centered within its corresponding sequence.

So far, the implementation of our algorithm for rendering curved text does not consider the convexity or concavity of the course of a street. If there is a convex part at the street course, we should shorten the distance between each two consecutive characters that lie at this part of the street. Analogously, for a concave part, we should enlarge the distance. We expect even more aesthetic labelings when we rotate the character that precedes the single character on the artificial segment; for instance, by the average value of the gradient of the preceding and the gradient of the artificial segment. Similarly, we can rotate the first character at the succeeding segment by the average value of the gradient of the succeeding and the gradient of the artificial segment.

5.4 Experiments

We have implemented the labeling algorithm of Section 5.2 in the version where the update of subsequences needs $\mathcal{O}(m^2)$ time. We also have implemented the visualization algorithms of Section 5.3. We used C++ with OpenSceneGraph 3.0[5.3]. We executed our experiments on a Windows 7 system with a 3.3-GHz AMD triple-core processor, 8 GB of RAM, and a GeForce GTX 460 graphics card, applying the Microsoft Visual Studio 2010 Ultimate compiler in 32-bit release mode.

For our experiments, we used a map from Geofabrik[5.4]. that provides OpenStreetMap data of the street network of Lower Franconia; a region in Southern Germany. As OpenSceneGraph for Windows is not able to handle such a large map, even if only a small part is visible, we extracted a map that shows a part of Würzburg; a town of 120,000 inhabitants. Our final street network has 620 polylines; see Figure 5.20.

As the computation time depends on the number of streets within the view, we executed our test for two different resolutions; one simulates the screen of a navigation system, the other one simulates the screen of a computer monitor; see Figure 5.21.

[5.3]`http://www.openscenegraph.org/`, accessed Nov. 24, 2013

[5.4]`http://download.geofabrik.de/europe/germany/bayern/unterfranken.html`, accessed May 20, 2014

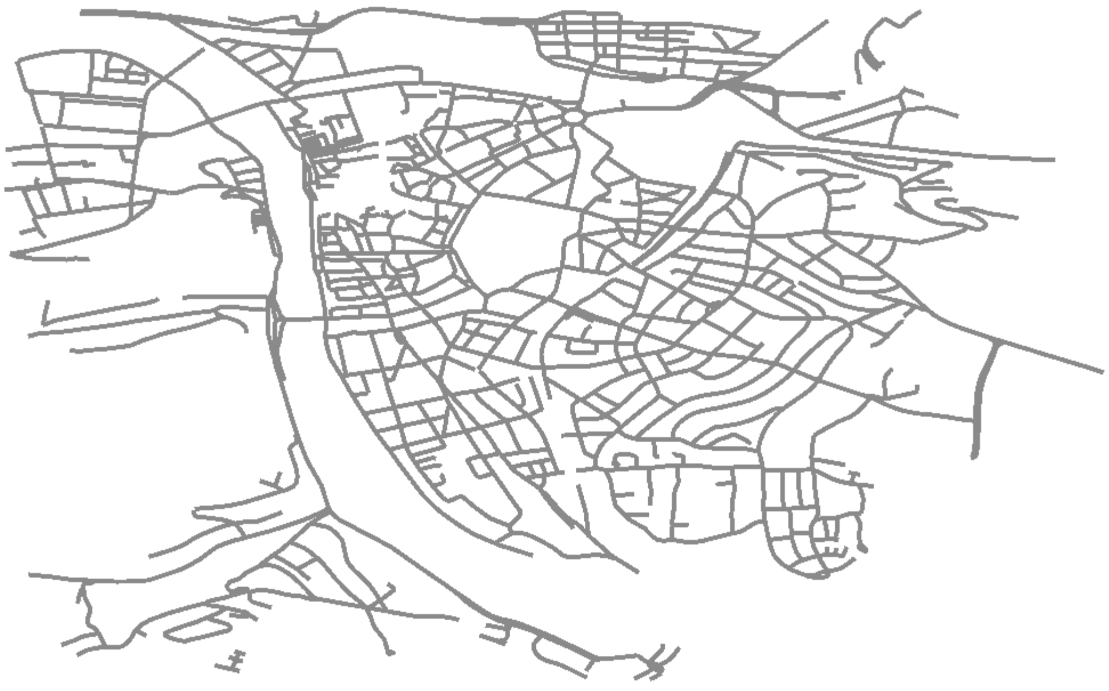


Figure 5.20: Street network that we used for our experiments.

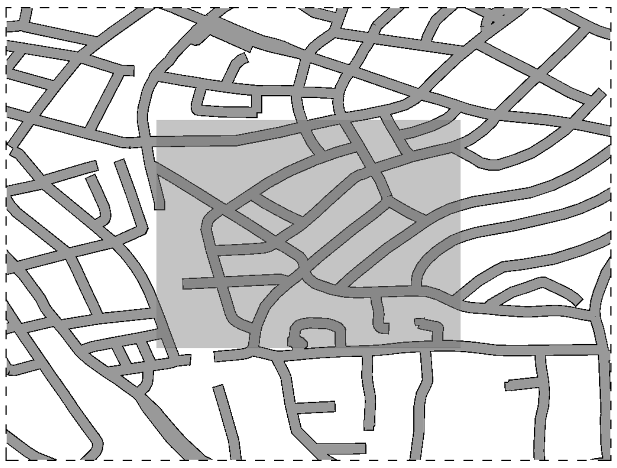


Figure 5.21: The complete map shows a view of our simulated monitor. The darker rectangle indicates the resolution of our simulated navigation system.

Furthermore, we implemented several camera paths. There are four *path classes* in which we only execute one interaction type, that is, we pan, zoom in, zoom out, or rotate. Each of these path classes contains five different paths at various scales. Each path lasts 14 seconds. At the largest scale, there are, on average, 5 labels on the navigation system display and 15 labels at the monitor; at the smallest scale, there are 15 labels on the the navigation system and 30 labels on the monitor. We used a 2D view. The fifth path class contains five multi-interaction paths. Each path mixes panning, zooming, and rotation operations for a total of 72 seconds. We distributed the time for each of these three interaction types (we considered zooming in and out as one type) equally at three different scales (whereas the largest and the smallest scale are the same as for the single interaction paths). We processed the multi-interaction path with a *camera angle* of 0° (2D view) and with an angle of 30° (see Figure 5.22).

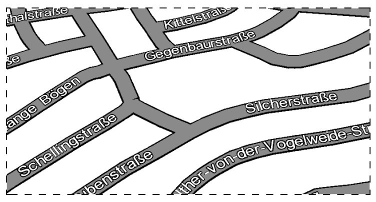


Figure 5.22: A camera angle of 30°.

For our experiments, we had to choose several parameters. After some testing, we obtained the following empirical values: we set the threshold for punishing too short segments to the average width over all characters; that is, $\tau = 0.57f$ where f is the current font size. Each junction incurs a cost of $X = 100{,}000$. For the evaluation, we set the value for enlarging the collection of visible segments to $v = |\ell|/2$, where ℓ is the label of the corresponding polyline. A sequence that intersects the view boundary costs $V = 100{,}000$. The maximum angle is $\alpha^\star = 90°$. We weight all criteria equally.

Results for Embedded Labels. Table 5.1 (embedded labels) gives an overview over the results of our experiments for rendering curved text; we discuss them in the remainder of this section. Figure 5.23, and Figures 5.28 to 5.31 at the end of this chapter depict some screenshots of maps that were visualized and labeled by our algorithms. As mentioned at the end of the introduction of this chapter, we also put a video online, showing the outcome of our algorithms[5.5].

In our tests for the map of Würzburg, for preprocessing the map and computing the values for the evaluation procedure, we needed less than one second each. On average, computing the initial labeling took us 0.08 seconds for the navigation system and 0.27 seconds for the larger monitor screen size.

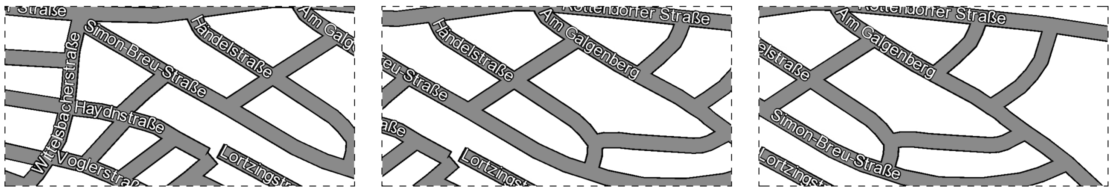
For each path, we measured the number of totally drawn frames as well as the total running time. By these two values, we computed the frame rate. Our implementation yields very good frame rates of 70–180 FPS for panning and rotating only and for the

[5.5]`http://lamut.informatik.uni-wuerzburg.de/dynalinelab.html`

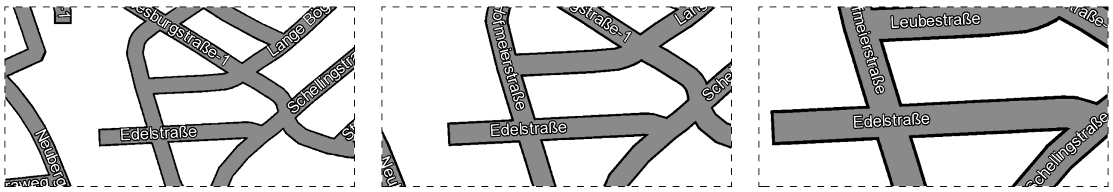
embedded labels							
perspec-tive	interaction type	screen size	∅f-rate [FPS]	all streets #visible	all streets %label.	long enough #visible	long enough %label.
2D	panning	GPS	176	26	34	11	78
		monitor	129	78	33	31	83
	zooming out	GPS	13	25	30	10	78
		monitor	4	56	36	24	84
	zooming in	GPS	12	25	33	10	83
		monitor	4	65	35	26	87
	rotation	GPS	183	22	40	11	81
		monitor	133	65	35	27	85
	multi-interaction	GPS	124	25	33	10	82
		monitor	89	67	35	27	86
3D	multi-interaction	GPS	111	28	37	12	85
		monitor	71	71	42	34	89

straight labels						
interaction type	screen size	∅frame rate [FPS]	all streets #visible	all streets %labeled	long enough #visible	long enough %labeled
panning	GPS	175	27	34	12	79
	monitor	142	79	33	32	83
zooming out	GPS	53	26	29	10	76
	monitor	21	63	35	26	84
zooming in	GPS	49	28	29	10	80
	monitor	20	69	34	27	87
rotation	GPS	175	27	34	12	79
	monitor	142	79	33	32	83
multi-interaction	GPS	140	25	32	10	81
	monitor	105	72	34	28	86

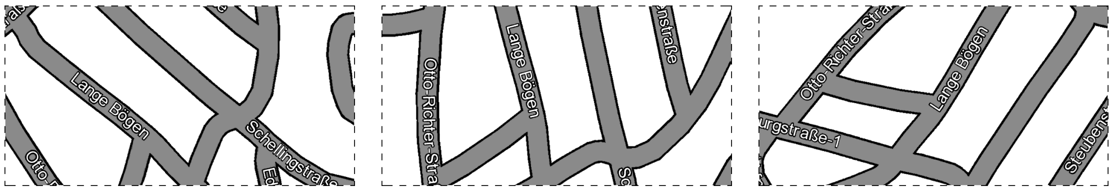
Table 5.1: Results of our experiments for curved labels in a map with a *2D* and a *3D* view and for straight labels in a map with a *2D* view. For both labeling styles, we tested camera paths that only allowed for one single *interaction type* and *multi-intercation* paths. We processed all the paths with a low number of streets within the view (like in navigation systems, *GPS* for short) and with a high number of streets (like at computers with *monitors*).



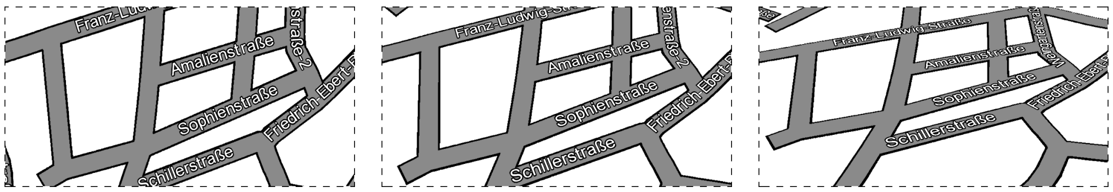
(a) From left to right: we move the view to the right. Initially, the label "Simon-Breu-Straße" is completely visible; then it partly leaves the view. Finally, it is placed at another position within the view and thus is completely visible again.



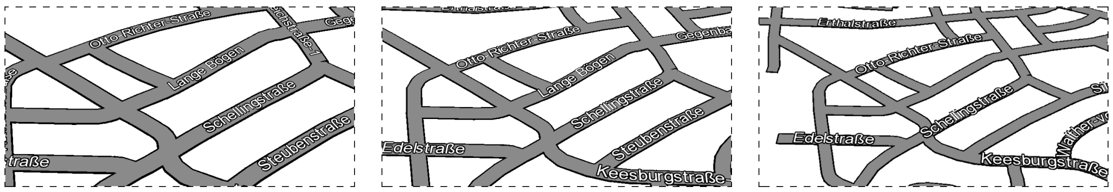
(b) From left to right: while zooming in, first the label "Hofmeierstraße" appears; then, the label "Leubestraße" shows up. Moreover, the label "Schellingstraße" is placed within the view again as the initial label for this street leaves the view completely.



(c) From left to right: we rotate the view clockwise; we correct the orientation of the label "Lange Bögen" (and other labels). New streets get visible in the corners of the view.



(d) From left to right: we change the camera angle. In the front and at the back, labels appear.



(e) From left to right: we zoom out; the view is perspective. Several streets lose their labels as they become to short; for example, "Lange Bögen" and "Steubenstraße". Some labels enter the view.

Figure 5.23: Screenshots of our program. We show each interaction type.

multi-interaction paths. At first sight, the frame rates of 4–13 FPS for zooming are unacceptable. On closer inspection, we spent most of the time for *rendering* the curved text. For panning and rotation operations, we only draw changes of the labeling. While zooming, we have to draw all the labels in each frame as, on the screen, the underlying streets change continuously. Switching off the routine for rendering curved text results in average frame rates of more than 155 FPS (independent of the interaction type).

At the multi-interaction path in the map with a 2D view, the frame rate of the navigation system is about 38% higher than for the monitor. Nevertheless, for the map in a 2D view, we reached average frame rates of 124 FPS for the navigation system and 90 FPS for the monitor. For maps in a 3D view, we reached an average frame rate of 111 FPS for the navigation system and 71 FPS for the monitor; thus, the frame rate for the navigation system is about 56% higher. The frame rate of the map in a 2D view outperforms the frame rate of the map in a 3D view by 11% when using the navigation system screen size and by 26% when using the monitor screen size.

Moreover, we determined the number of *labeled* streets, the number of *all* visible streets, and the number of streets that are actually *long enough* to host their labels. Our algorithm labels only about 34% of all streets, but it labels about 80% of the streets that are long enough. Due to the rather small angle of the 3D view, the number of visible labels in the map with a 3D view is only slightly larger than the number of visible labels in the map with a 2D view. Recall that, due to the NP-hardness of our problem, we only developed a heuristic. As the heuristic places labels incrementally, local decisions might prevent solutions that are globally better; see Figure 5.24. Note, however, that even in optimal solutions there can be unlabeled streets.

As in general a frame rate of 24 FPS is qualified as fluid, we conclude that our algorithm is highly real-time capable in most situations. When zooming, the rendering of the curved text is too slow. This should be improved.

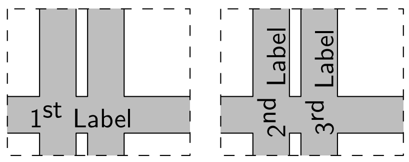


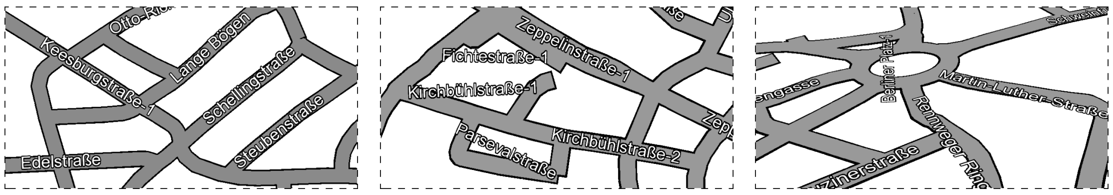
Figure 5.24: If the label of the horizontal street is considered first, the labels of the two vertical streets possibly cannot be placed.

Results for Straight Labels. It is quite hard to compare our results with the results of the two other existing algorithms for labeling line features in interactive maps as they only place straight labels—and rendering curved text is the bottleneck of our program. We first tested if our algorithm yields better frame rates if we label maps where streets have quite straight courses. The idea is that, in such street networks, our rendering algorithm can often place the complete street name at once. We come to the conclusion, that, for labeling Manhattan (United States), the frame rate only increases by about

1 FPS for zooming operations (independently of the screen size) whereas the number of placed labels in Manhattan almost equals the number of placed labels in Würzburg. On that account, we adapted our algorithm such that it is also able to place straight labels. We only exchanged the rendering routine: instead of placing substring by substring, we place a label such that it is centered on the line between the start and the end point of the corresponding sequence; see Figure 5.25. We executed the same camera paths as before but the multi-interaction paths in the 3D view. We show the results in Table 5.1 (straight labels). The frame rates for panning and rotation operations do not differ much. The frame rates for the multi-interaction paths increase slightly by about 16 FPS (13%). The average frame rate for zooming increases by about 40 FPS for the navigation system and by about 17 FPS for the monitor screen size.

To compare, Maass and Döllner [MD07] achieve frame rates of 17–22 FPS using an 2.93-GHz Intel Core 2 Duo processor with 2 GB of RAM and a GeForce 7950 GT graphics card. Vaaraniemi et al. [VTW12] compute the layout for 512 map objects of various types using an 1.6-GHz Intel Core 2 Duo processor with 4 GB of RAM and a GeForce 8600M GT graphics card with 256 MB of RAM. With parallel computation on the GPU, their algorithm needs about 5.5 milliseconds (which corresponds to 180 FPS) for once computing all label positions but *without* any rendering.

Clearly, the frame rate depends on the number of visible streets and on the hardware. Moreover, we know nothing about the executed camera paths; more precisely, we do not know the frame rates for zooming operations.



(a) good label–object association (b) poor label–object association; 2D view (c) poor label–object association; perspective view

Figure 5.25: Labelings with straight labels computed by our evaluation algorithm.

5.5 Extensions

In this section, we give some ideas to improve our algorithm. We describe how to handle the weighted case, criteria that might make the labeling looking better, and a method that probably speeds up our approach. We only explain the concepts while omitting details.

Weighted Case. Assume that every polyline π (or street) comes with a weight $w(\pi)$. The weight $w(\ell)$ of a label ℓ is always the same as the weight of its corresponding street. The higher $w(\ell)$, the more important it is to place ℓ. We still stick to the idea that labels should not move as moving labels distract the user. For that reason, there is only *one* exceptional situation in which we permit to move a label: consider a polyline π that only has candidates $\ell_1, \dots, \ell_i, \dots \ell_k$ that pass junctions which are already occupied by labels of other streets. Let L_i be the set of labels which are overlapped if we place the label of π at ℓ_i. Next, for each L_i, compute the sum $\sum_{l \in L_i} w(l)$ of the weights of every label l in L_i. The sum reflects the cost for making space for the label of π. Let L^* be the set that yields the lowest costs. If $w(\pi) \leq \sum_{l \in L^*} w(l)$, we do nothing. Otherwise, we remove all labels in L^* from the map and place the label of π. We apply our evaluation routine to every street that lost its label due to the label exchange. For our decision, we could also take into account whether a removed label can be replaced or not. If we can replace it, it has a lower cost (maybe zero) than a label that cannot be replaced.

One could argue that this approach does not satisfy common rules for map productions; for example, several unimportant side roads could prevent placing the label of the main street as the sum of several low weights exceeds the weight of the main street. If we choose a suitable distribution of the weights, this can not happen, though. For instance, let ω be the weight of a side street. Then set the weight of the main street to 100ω.

Centering. We first have to discuss what *centering* means in this case. In static maps, if there is only one single label for a street, it is surely desirable to place the label at the center of its street. In a dynamic scenario, however, this approach may lead to a label position that lies partly or even completely outside the view. For that reason, we aim to place the label at the center of the visible part of the street. We only need to sum an additional cost while we evaluate a sequence; the smaller the distance to the center of the visible part of the street, the lower the cost (see Figure 5.26). A rather flat function indicates that it is less important to center the label; using a steep function, it is important. If it is extremely important to center the label, we also can multiply the additional cost.

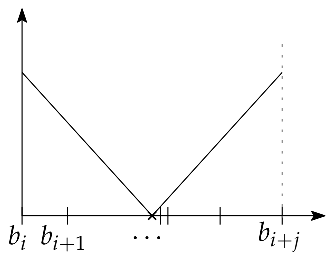


Figure 5.26: Example for an additional cost in order to center a label at the visible part of its street. The bend b_i is either the start point of the street or the point where the street enters the view. Analogously, b_{i+j} is the end point of the street or the leaving point.

Multiple Labels. Imhof [Imh75] imposes that long polylines should be labeled multiple times. More precisely, he suggests repeating labels whenever lines are visually not distinguishable, that is, where the label–object association could be violated (for instance, at forks). We weaken the requirement and just aim for repeating labels. With this, we also solve the problem of missing street names in the case that a street enters and leaves the view multiple times.

For placing multiple labels of the same street, we consider the entire street when the street gets into the view. Certainly, the first label should be visible as soon as possible. For that reason, we immediately apply our evaluation algorithm for the visible part of the street. The next label should neither be too near to nor to far away from the first one. We model this behavior with the help of an additional cost; see Figure 5.27. We block the part (or a slightly larger part) of the street where we placed the first label. To this end, we set an infinite cost for the occupied region. At the boundaries of this region, we set a finite value that decreases continuously; the further away the position of the next label, the lower the additional cost. If we also want to define a maximum distance of the two labels (for instance, the height of the view is a reasonable value), we raise the cost after a specified distance. We apply the same routine for placing any further label.

If we allow for multiple labels for one street, we need a collision detection in order to avoid overlapping labels if the user zooms out.

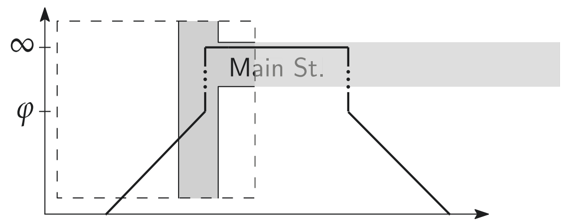


Figure 5.27: Example for an additional cost for placing several labels of the same street. The current view is shown on the left: a new street appears on the right. The position of the first label indicates an infinite cost; this area is blocked for any other label. The value φ indicates the transition of finite and infinite costs.

Inside the View. Hitherto, if we evaluate a sequence that intersects the view boundary, we include a constant cost V in the total cost of the sequence. Alternatively, we can measure how much of the sequence is visible; we scale V accordingly and include the scaled value in the total cost of the sequence. This is, however, a only heuristic that might estimate inaccurately. With somewhat more computation time we also could test how much of the *label* is visible.

Running Time. Of course, there are common ways to improve the running time of an implementation by, for example, using multithreading (one processing unit) or parallel processing of at least two processing units. More important for us is to reduce the number of useless tests and computations.

When using perspective labels, we can predict values for how far users can pan to any direction and how far they can zoom until a street has to be labeled or loses its label. Tilting corresponds to a panning operation; therefore, we do not need any special predictions. Further, we do not compute the values for rotation operations as these values change permanently while the user pans or zooms and changing the label orientation only needs constant time. Now, by means of these values, the only test we have to execute in each frame is if we reached such a value. If so, we have to update the labeling. We also have to update the values for the predictions at appropriate points in time. With this strategy we spent somewhat more computation time and space but save repeated tests for unlabeled streets and with that the repeated application of our algorithm.

5.6 Concluding Remarks

We presented a real-time algorithm that dynamically attaches curved labels to line objects in interactive maps while trying to label as many different streets as possible and selecting nice-looking label positions. Our algorithm implicitly considers all possible label positions of the visible part of a street by evaluating only a discrete number of positions explicitly. Our algorithm directly reacts to user interactions. In tests on real-world data, our algorithm reached average frame rates of more than 90 FPS on multi-interaction paths and labeled about 80% of the streets that are long enough to host their labels.

For the future, we should conduct a user study in order to verify the aesthetics and usefulness of our labelings. Ideally, this should be done in cooperation with psychologist. Concerning the implementation of our algorithm, it would be interesting to adapt it such that it can handle the weighted case and place multiple labels for the same street. Moreover, we should improve the computation time for the algorithm rendering curved labels. We refrain from using the four-slider model as the one-slider model already causes much label movement.

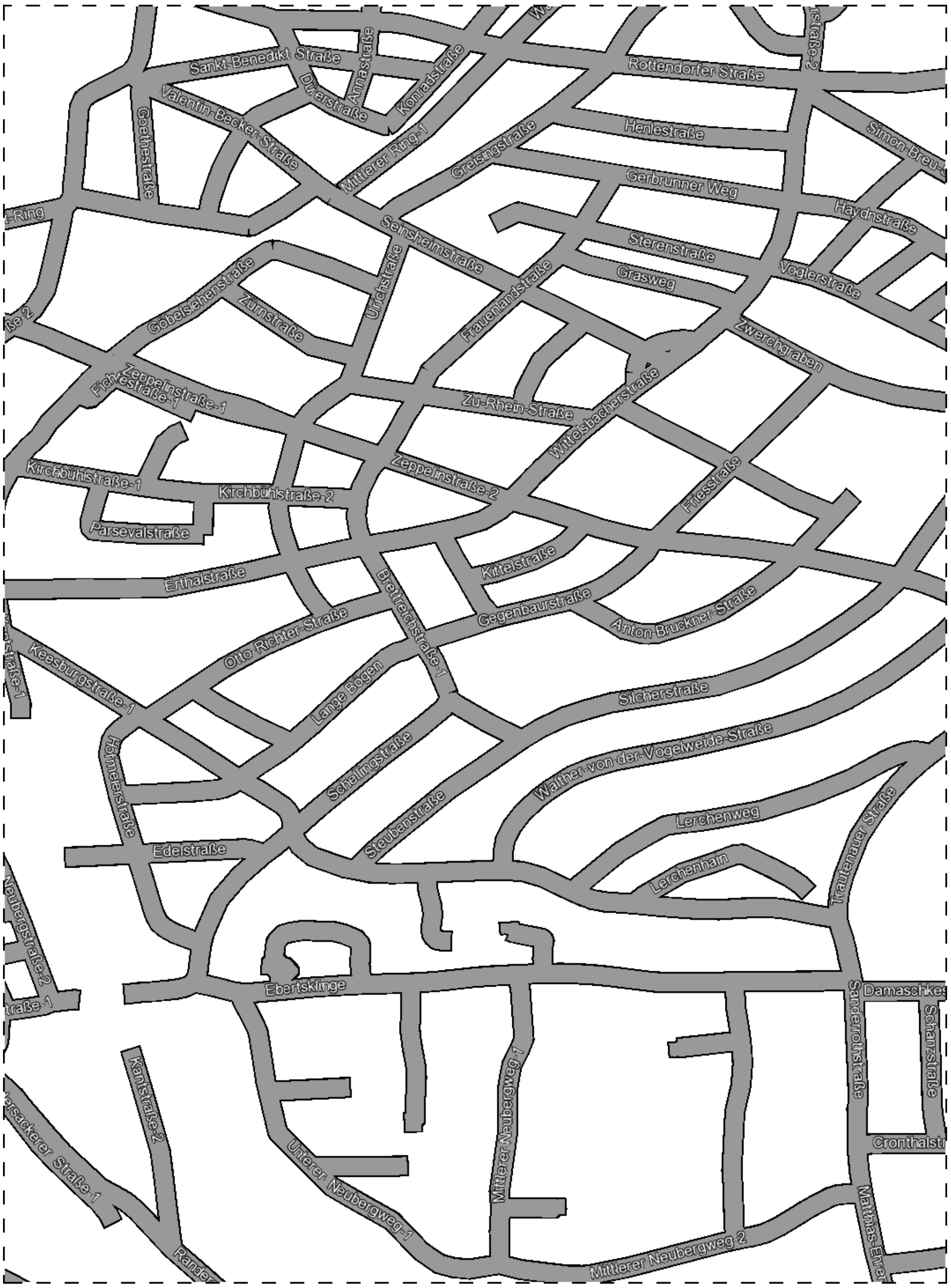


Figure 5.28: A map of Würzburg (Germany) in a 2D view, computed by our algorithms.

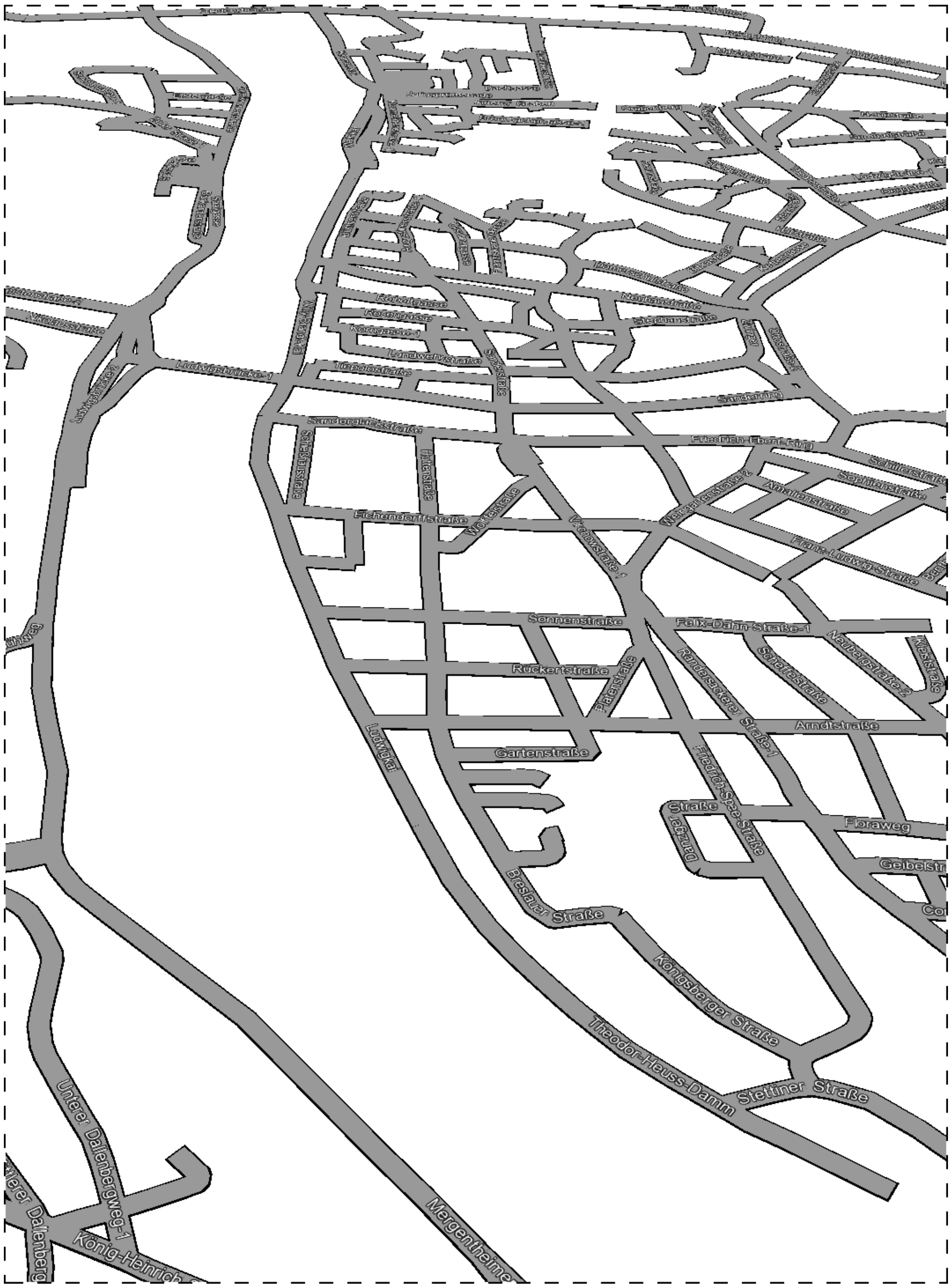


Figure 5.29: A map of Würzburg (Germany) in a 3D view, computed by our algorithms.

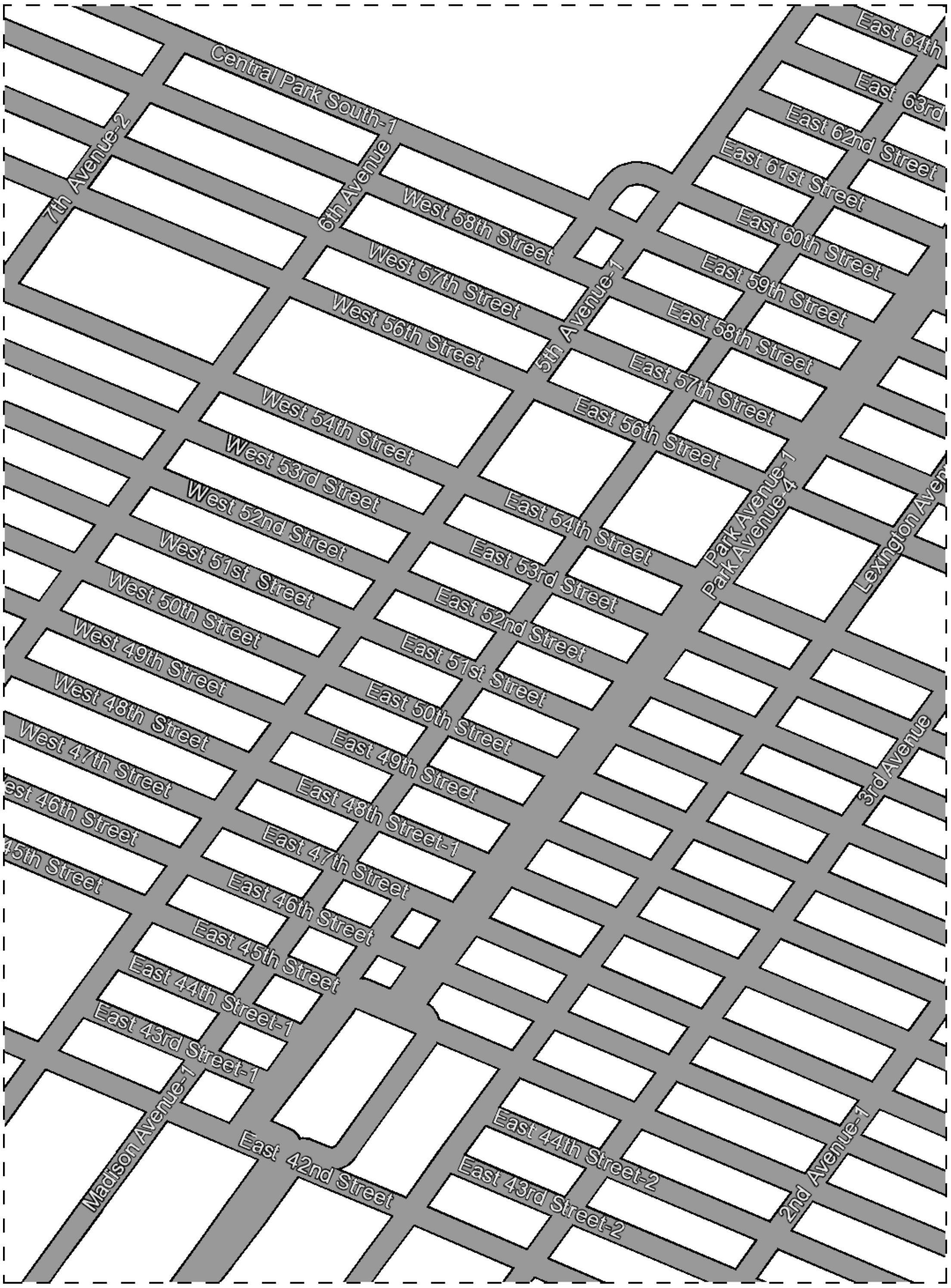


Figure 5.30: A map of Manhattan (U.S.) in a 2D view, computed by our algorithms.

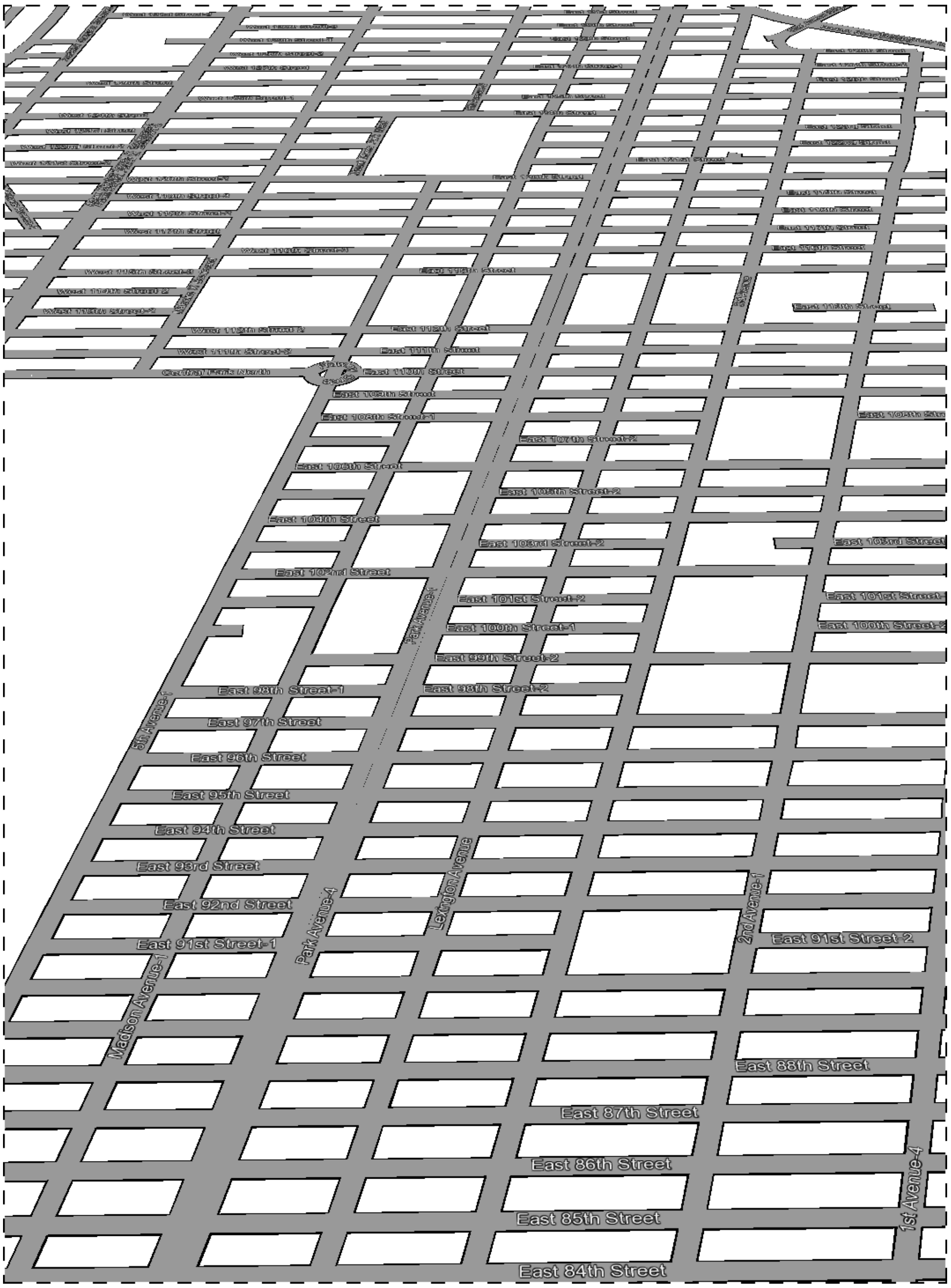


Figure 5.31: A map of Manhattan (U.S.) in a 3D view, computed by our algorithms.

Chapter 6

Labeling Streets Along a Route with Billboards

In the previous chapter, we studied the problem of attaching embedded labels to streets with spatial extend aiming for maximizing the number of different streets that are labeled while prohibiting label–label overlaps at junctions. In this chapter, we consider the problem of attaching billboards to every street along a user's route. We consequently need a reference point where to place the billboard on each street. In our case, a *billboard* consists of (i) a *label*, that is, a rectangle that is oriented towards the user and holds the *label text*, and (ii) a *leader* that connects the point to be labeled with the label. The leader of a billboard can be a line or a more complex object such as a triangle or an arrow. We use vertical lines that anchor their labels at the center of the labels' bottom edges.

Mobile devices with interactive maps commonly provide a *map mode* in which users can freely interact (literally, users travel with their fingers on the map). Additionally, the same devices usually offer a *navigation mode*, a route planner, that leads users from their current locations to specified destinations. Sometimes, it is even possible to interact with a map that is in navigation mode. When the navigational device leads the user to a destination, the route which the user has to follow is usually highlighted. We call this route and the corresponding streets *active*. Accordingly, we refer to streets that are not contained in the active route as *inactive*. By using billboards instead of embedded labels for the active route, we highlight active streets. Moreover, horizontally-written text can be read faster than rotated text [Tin72, KN85, WB05] and billboards are judged to be better legible than embedded or straight labels [VTW12] (see the related-work section of Chapter 2). We thus improve the legibility of those labels that are, at any given time, most important for the user.

In this chapter, we develop a real-time algorithm that attaches billboards to streets in interactive maps that are in navigation mode. We assume that the future course of the route within the currently visible part of the map is known or well predicted. Note that the route can change quite spontaneously, for example, if the user makes a wrong turn. We aim for an algorithm that computes an aesthetic labeling for new routes very fast, that is, in real time. As we want to label every street on the visible part of the route, we permit label–label overlaps but try to keep the overlapped area small. To this end, we dynamically change the lengths of the leaders in each frame. Nevertheless, we require a smooth movement of the labels. On that account, we limit the change of the length of a leader per frame. This obviously satisfies the consistency criteria of Been et al. [BDY06]:

if we choose the limit suitable, labels do not jump. (We remark that, in this scenario, billboards cannot flicker. We only remove a billboard shortly before the user passes it; if the route does not change, this label will not be placed again.)

For labeling inactive streets, we can use our algorithm for embedding labels into streets (see Chapter 5). Indeed, we have implemented the combination of attaching billboards to active streets and attaching embedded labels to inactive streets. We will present the results in the following chapter. Although, in this chapter, we only study the problem of labeling streets, that is, line features, with billboards, our algorithm can also be used for labeling point features (without any changes) or area features (assuming that we are given a suitable point in each area feature).

Our Model. We consider a dynamic scenario where the user follows a route in an interactive map in navigation mode. In our scenario, the user can continuously pan, zoom, rotate, and tilt a perspective view. At any given time t_i, or frame, the user can see a trapezoidal region R_i of the map. In this chapter, we can use the same formal description for the region R_i as for the embedded labeling (see Chapter 5).

Further, for the model of our problem DYNAROUTELAB, we lean on the physical principle of a thermodynamic equilibrium. We assume that in each frame there might be a label–label overlap because either a new label has come into the view, an overlap of the preceding frame has not been completely solved, or a solved overlap from the preceding frame has caused another overlap. Each label–label overlap induces a force F_{overlap}. We additionally define a desired leader length. A leader that is too long or too short induces a force F_{leader}. We correct the leader lengths from one frame to the next. We can translate the goal of establishing an overlap-free labeling in one frame to the goal of minimizing the acting force

$$F = \sum |F_{\text{overlap}}| + \sum |F_{\text{leader}}|$$

in that frame. We aim for minimizing each individual force as forces of the same billboard might cancel each other out even though there is still a label–label overlap. This happens, for example, if label A pushes label B downwards but B is also pushed upwards by the force from having a too short leader.

With the application of a physical model, we expect the movements of labels to look natural—as if they were subjected to physical laws.

Our Contribution. We present a force-directed algorithm for dynamically attaching billboards to streets on a user's route in an interactive map in navigation mode (see Section 6.2). Throughout the navigation, the algorithm maintains an aesthetic and useful labeling; it uses repelling forces for resolving overlaps and both attracting and repelling forces for keeping leaders close to their desired length. All label movements are smooth. Tests of the implementation of the algorithm on real-world data show that our algorithm yields frame rates of more than 420 FPS, that is, our algorithm is real-time capable (see Section 6.3). Moreover, compared to an algorithm with a fixed leader length, our algorithm drastically reduces the overlapped area. To be exact, the overlap caused by our algorithm is less than 2% of the overlap caused by the algorithm with a fixed leader length. We conclude this chapter by proposing some improvements for our algorithm, for example, to name just a few, how to deal with leaders of arbitrary directions, applying our algorithm to maps in the map mode, and we discuss the difficulties when placing several billboards per street (see Section 6.4). As in the previous chapters, we provide a video that shows our algorithm in action[6.1].

6.1 Related Work

As we already know, in his seminal work, the Swiss cartographer Imhof [Imh75] establishes many rules for good label placement whereas his two most important rules are that labels should be legible and always yield a correct label–object association. With our algorithm for labeling streets with billboards, we fulfill these rules since we align labels horizontally, we connect the label and the reference point by a leader, and we avoid overlaps. He also states that a label should reflect its object's importance. In our setting, where we want to label streets on the active route, Imhof's rule is automatically fulfilled since our perspective view draws the closer (and, hence, in that moment more important) labels in the foreground larger than the distant (and less important) labels in the background. Further, Imhof points out that labels should occlude the map background as little as possible. Currently, we do not satisfy this rule. It can, however, be achieved by making the bounding box of the label semi-transparent (what only slightly diminishes the legibility of the label [HV96]; see related-work section of Chapter 2). In order to avoid label clusters, Imhof suggests carefully selecting the objects to be labeled. In our navigation-mode scenario, we simply select the next n streets on the route to be labeled. Among these, we show all labels that fall into the current view. We avoid clusters by changing the leader lengths such that there is an additional distance between each pair of billboards.

For drawing graphs aesthetically, Eades [Ead84] introduces an algorithm which is based on a physical model using forces. He considers a drawing aesthetic if the edges of the graph have similar lengths and the graph is as symmetric as possible. To this end, the vertices of the graph may move in any direction. Adjacent vertices are supposed to keep a certain distance from each other, non-adjacent vertices repel each other. In our model, by contrast, the reference point of a label is fixed and the point where leader and label touch can move only vertically. Similar to the edges in Eades' approach, our leaders try to have a certain length. The author states that he does not use Hooke's law (as we do) but a logarithmic function to obtain the edge lengths because a logarithmic function works better for vertices that are far apart. As typical for force-directed approaches, Eades' algorithm computes the forces, and thus the new positions of the vertices, several times. Our approach is also iterative: in each frame, we recompute the lengths of the leaders if the corresponding labels overlap or the leaders are not at their desired lengths.

[6.1] `http://lamut.informatik.uni-wuerzburg.de/dynaroutelab.html`

Force-directed algorithms for computing labelings have also been considered for static maps. Hirsch [Hir82] introduce the first such algorithm. He uses a labeling model that is similar to the four-slider model (see Figure 4.1, page 53). The difference is that the label edges touch a circle that is centered at the point feature to be labeled (instead the feature itself). The aim is to minimize the overlapped area. Hirsch repeatedly computes a vector, or rather a force, for each label–label overlap, sums up the forces of each label, and moves the labels accordingly along the circle. The primary drawback of force-directed algorithms (which is also present in Hirsch's algorithm) is that solutions are easily caught in a local minimum. In order to overcome this issue, Ebner et al. [EKW05] combine a force-directed algorithm with the technique of *simulated annealing* [KGV83], that is, with a certain probability, the total label–label overlap gets worse from one iteration to the next. With each new iteration, this probability shrinks.

Table 6.1 gives an overview about our algorithm and some of the related algorithms for labeling interactive maps. We discuss the algorithms in the following.

	dim.	interaction types	history	mode
Vaaraniemi et al. [VTW12]	3D	pan, zoom, roate, 3D	considered	map
Maass & Döllner [MD06]	3D	*unknown*	by workaround	map
Gemsa et al. [GNN13]	2D	pan, rotate	considered	GPS
our approach	3D	pan, zoom, rotate, 3D	considered	GPS

	computation time	objects	technique
Vaaraniemi et al. [VTW12]	5.5ms per update	all (by points)	force-based
Maass & Döllner [MD06]	”real time“	points	greedy
Gemsa et al. [GNN13]	seconds to minutes	points	ILP, greedy
our approach	> 420 FPS	streets (by points)	force-based

Table 6.1: Our approach compared to some related work. We use *dim.* as abbreviation for dimension, *GPS* for navigation, and *ILP* for integer linear program.

Vaaraniemi et al. [VTW12] give a force-directed algorithm that is, to some extent, similar to ours. Their algorithm labels point and area features horizontally. Depending on the current perspective, a street label is either placed horizontally or straight. Around each label, the authors define a buffer zone into which no label may be placed or moved. In contrast to our approach, their algorithm operates in what we call map mode and they allow any leader direction. The main difference between the two approaches is that while we always display all labels that fall into the view, Vaaraniemi et al. remove labels in two situations, namely if a label moves too fast or if a label is overlapped such that the forces acting on it cancel each other. While we label only the active streets, Vaaraniemi et al. label all types of objects within the view. As they resolve overlaps by moving and selecting labels, we expect a lot of changes on the screen which may be distracting (for example, for a car driver using a navigational device). In terms of speed, Vaaraniemi et al. report that their algorithm computes the layout for 512 objects within 5.5 milliseconds (this corresponds to 180 FPS *without* rendering.)

Gemsa et al. [GNN13] investigate the offline version of a point-labeling selection problem (in a 2D view) with respect to an active route. While driving along the route, the view is panned and translated. The object that indicates the user, the pointer, stays at a constant position on the screen. As common in navigational devices, the view automatically changes such that they always drive upwards. The authors assume that the entire active route is given in advance and fixed (while it may change in our case). They do not use leaders but they assume that each label has a fixed position relative to the point feature it labels. While passing the route (without preventing overlaps), each label may be visible during several intervals. In order to reduce flickering, for each of these intervals, Gemsa et al. [GNN13] allow for selecting a *connected* (sub)interval in which the corresponding label is finally visible. The authors aim for an overlap-free labeling for the entire route that maximizes the total length over all selected (sub)intervals. Note that this problem is an ARO (see Chapter 3). The authors show that their problem is NP-complete and also $\mathcal{W}[1]$-hard, that is, there is probably no *fixed-parameter algorithm* (broadly speaking, fixed-parameter algorithms solve some instances of an NP-hard problem efficiently). Nevertheless, the authors present an integer linear program that solves the problem optimally but in exponential time. They test their approach on 1,000 active routes at three different scales. On average, they need less than a second for optimizing the labeling of routes with about 162 labels and less than six seconds for routes with about 313 labels. A few routes, however, took them several minutes. For the case that labels are unit squares, the authors give an approximation algorithm with factor 8 that runs in polynomial time. The authors also restrict their problem to the more practical case that, while following a route, never more than k labels (of arbitrary size) are shown simultaneously; they give an approximation algorithm with factor $\min\{3 + k, 11\}$ that runs ins polynomial time. For the approximation algorithms, the authors do not provide any experimental test results.

Maass and Döllner [MD06] place billboards in interactive 3D virtual environments. They require that the further away the labeled object is from the user, the smaller the label and the higher the leader. Their algorithm subdivides the view into a grid and places labels incrementally. Each placed label blocks several surrounding grid cells for other labels. The algorithm does not consider the history of the labeling. In order to avoid label jumps that might confuse the user, the labeling does not change while the user interacts with the map. If the user stops interacting, labels smoothly move to their new positions. In contrast, we immediately react to a user interaction while considering the labeling of the preceding frame.

We observe that billboards are usually used for boundary labeling where labels are placed outside the actual scene. Gemsa et al. [GHN11] use this technique to label point features in panorama images. The authors assume that the point features lie below a horizontal line, the *horizon*. The labels are placed in multiple rows above the horizon (the *sky*) where they are assumed to not occlude interesting image features. Every label is a unit-height rectangle that, as in our labeling model, is connected with the corresponding reference point via a vertical leader. Labels are not allowed to overlap other labels or intersect leaders of other labels but they are allowed to slide horizontally. The authors present an efficient algorithm that places labels for all point features in a minimum number of rows. They show that a feasible solution always exists but, in the worst case, at most two labels can be placed in each row. Consequently, the space consumption in the vertical direction can be very large. To overcome this issue, the authors also develop efficient algorithms for problem variants in which the available space for labels is limited and the aim is to label a set of points of maximum size or maximum total weight.

6.2 Algorithm

In this section, we propose a simple force-directed algorithm for solving DYNAROUTE-LAB heuristically. All in all, we need several auxiliary algorithms, for instance, setting up, reading, and routing through a street map. We will not study these algorithms in depth, though.

Consider a street map which contains an active route from a starting location A to a destination location B. The active route is a sequence of *street polylines* where each street polyline is only a subsection of the entire street in the sense that one traverses a street for only so long as to reach the junction that leads to the next street. We use the terms street polyline and street interchangeably. The route is traversed by a *pointer* π which represents the user. A pointer typically is a triangle or a vehicle; see Figure 6.1. The camera is placed at some distance behind the pointer.

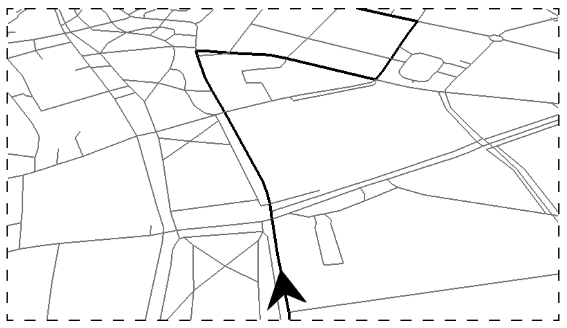


Figure 6.1: A street network with a route (thicker) and a pointer (triangle).

Our aim is to label each street of a route while dynamically preventing labels from overlapping each other. To this end, we place billboards in world space, each above the reference point of its street. For the sake of simplicity, we assume that each street—regardless of its length—has only one reference point, namely at the midpoint of its polyline. (As there are things to respect when placing several billboards per street in a 3D view, we discuss this issue in Section 6.4.) Recall that, to connect a label with its reference point, we use a vertical leader whose length can dynamically vary. We denote the desired height of a leader, the *default leader height*, by h_0. Obviously, our labels can move only vertically, namely by extending or contracting the leaders. We set the minimum height of a leader to zero, that is, a label always lies above its reference point such that the label at least touches it. To simplify further discussion, let us consider each billboard b_i as a complete entity whereas i denotes the i-th street of the route. We denote the label of b_i by ℓ_i and the actual height of the corresponding leader at any given time t is $h_i(t)$.

Given the route R, let $N := |R|$ denote the number of streets along the route. It makes little sense to display each annotation for every street in the route at the same time as the user is primarily interested in the next few streets ahead. Therefore, we limit the number of *placed* billboards at any time to a constant $n \leq N$. Let $I = \langle b_l, \ldots, b_m \rangle$ denote the queue of currently placed billboards; we have $|I| \leq n$. (We have $|I| < n$ if the remaining part of R consists of fewer than n streets.) When the distance of the pointer π to the reference point q_l of billboard b_l falls below a threshold ε, then the billboard b_l is dequeued from I and the billboard b_{m+1} (if it exists) is enqueued. Note that the number n of placed billboards is sometimes larger than the number of billboards within the view. We also place billboards that lie outside the view, in addition to the visible billboards, for two reasons. First, we can omit checking if a reference point is about to enter the view. With this, we save computation time. Second, when billboards do enter the view, they do not disturb the visible labeling much as they have already been considered by the algorithm and thus they do not cause much overlap or movement.

6.2.1 Force-Directed Approach

In our force-directed algorithm, on each label, forces are exerted by other labels and by the leader of the label. The forces cause the label to move such that the *total force* is minimized. The leader acts as a spring, keeping the label close to its reference point while, simultaneously, labels repel each other—much in the same way that same-pole magnets do. The total force is mapped to a change in leader height. In order to prevent a label from oscillating strongly between two other labels, the force is scaled by a *temperature* which is reduced when the total force changes its sign from one iteration to the next.

While the billboards themselves live in the 3D world space, we see them in a projection into the 2D screen space. The screen-space representation of a label is a rectangle (note that the leader is not included). We determine if two labels overlap by inspecting whether their screen-space projections overlap. (In Chapter 4, we have already pointed out that we cannot determine overlaps in world space.)

One last note before we move on: algorithms in this chapter are *frame-based*. In each new frame, the pointer and the camera may have moved, billboards may be seen from a different perspective, labels may have been moved up or down by their leaders, auxiliary algorithms take the new data into account, and the force-directed algorithm runs. Frames may coincide with rendered frames or they may be timer-based. The only requirement is that the changes of the leader heights are reflected in the screen-space projection in the next frame. Since the information the algorithms need is primarily for the current frame, for notions of frame or time-bound values, we omit the time t; for example, we simply use h_i instead of $h_i(t)$.

In the following, we discuss how to compute various forces and how they change the leader heights.

6.2.2 Spring Force

The leader of a billboard is modeled as a simple tension spring that has a default height of h_0. The spring can undergo extension as well as contraction, that is, negative extension. The *spring force* F_i^{s} of a billboard b_i is given by Hooke's law and is simply a spring constant k multiplied with the leader extension:

$$F_i^{\mathrm{s}} := -k \cdot (h_i - h_0) . \tag{6.1}$$

Due to the way the spring force flows into the total force acting upon a label, the main effect k has is to affect the speed at which the corresponding leaders return to their default height: the larger k, the stronger the force.

6.2.3 Aggregate Repulsive Force

Principally, the *aggregate repulsive force* F_i^{r} of label ℓ_i consists of the sum of all *repulsive forces* $r(i,j)$ between label ℓ_i and every other placed label ℓ_j:

$$F_i^{\mathrm{r}} := \rho \cdot \sum_{j \in I \setminus \{i\}} r(i,j). \tag{6.2}$$

As we will see in Section 6.2.4, this is a slight simplification but, for initial understanding, it is sufficient. The constant ρ serves to weight the aggregate repulsive force. The function r relies on several concepts, we introduce next: the location of a label to any other label, the distance function, and the interplay between labels.

Location Function. Let $\mu(i)$ be the midpoint of the projection of ℓ_i and let $\mu_y(i)$ be the y-coordinate of $\mu(i)$. Let the *location function* $\sigma(i,j)$ of label ℓ_i denote whether ℓ_i lies above or below another label ℓ_j, $i \neq j$:

$$\sigma(i,j) := \begin{cases} -1 & \text{if } \mu_y(i) < \mu_y(j) \text{ and} \\ 1 & \text{otherwise.} \end{cases}$$

Indeed, we can categorize each ℓ_j in one of three categories with regard to ℓ_i: either the midpoint of ℓ_j is below, above, or at the same height as the midpoint of ℓ_i. The location function σ, however, handles only two cases. One might ask: might not two labels, in case of midpoint equality, move upwards at the same rate and thus, over time, arbitrarily far from their reference points? We think that this cannot happen because multiple factors affect the change of the leader heights and these factors cannot all be the same, unless in the case $\ell_i = \ell_j$ that we exclude. Hence, one of the two labels shall travel further in one frame; in the next frame, the midpoints have different y-coordinates.

Distance Function. We can model the repulsive force between two labels as a function of their distance to one another. This is an attractive model as it is easy to reason about but it leaves us with the problem of defining a suitable *distance function* δ.

We observe that, if two labels do not share a common x-coordinate, they cannot affect each other by changing the leader heights. In our distance function, we model this independency by setting the distance of such two labels to ∞.

What if the labels share a common y-coordinate? Typically, the distance between two objects is measured starting from zero when the objects are right next to each other. To deal with this, we define that one distance unit equals the height of a label in world-space coordinates. Let $\omega(i,j) \in [0,1]$ denote the screen-space area that ℓ_i and ℓ_j overlap relative to the area of ℓ_i. For example, if $\omega(i,j) = 1$, ℓ_i is completely covered by ℓ_j. Note that ω is asymmetric. Second, in the case that ℓ_i and ℓ_j do not overlap (but share a common x-coordinate), let $\gamma(i,j)$ denote the distance between ℓ_i and ℓ_j at the y-axis. This leads to the following definition of δ:

$$\delta(i,j) := \begin{cases} \infty & \text{if } \ell_i \text{ and } \ell_j \text{ are independent,} \\ 1 - \omega(i,j) & \text{if } \omega(i,j) > 0 \text{, and} \\ 1 + \gamma(i,j)/h_i & \text{otherwise.} \end{cases}$$

Note that this definition is inconsistent because $\delta \in [0,1]$ is a measure of area and $\delta > 1$ is a measure of height. It would make little sense to define the distance of two labels at the y-axis as an percentage value. On the other hand, we consider it more useful to depend the force on the overlapped area than on the overlap in one axis. For example, labels with a large height can induce a large overlap whereas indeed only the boundary is overlapped; often, such an overlap does not disturb the user.

Final Repulsive Force. We give the algorithm that computes the final repulsive force $r(i,j)$, which ℓ_j exerts upon ℓ_i, in Algorithm 6.1. We describe the idea in the following.

When a label ℓ_i is completely covered by another label ℓ_j, that is, when $\delta = 0$, we restrict the repulsive force $r(i,j)$ to a constant F_{limit} in order to avoid that ℓ_i jumps. Consequently, F_{limit} defines the maximum-allowed moving distance. When $\delta = 1$, we let the force $r(i,j)$ equal the spring force F_i^s. For $\delta < 1$, the force $r(i,j)$ grows quadratically. For $\delta \in [1,2]$, the repulsive force $r(i,j)$ grows linearly. Finally, we define that labels with $\delta > 2$ do not affect each other—their distance at the y-axis is large enough. If the labels should come closer, then, at some point in time, $\delta \leq 2$ will hold.

We show the graph that corresponds to the repulsive force r in Figure 6.2. It is monotonic and continuous. Alternatively, we could define the force such that it approaches the maximum force F_{limit} when $\delta \to 0$. This would, however, only provide questionable benefit while requiring higher computation time.

The definition of the force r has the fortunate property that the force acting upwards on label ℓ_i from a lower label ℓ_j will equal the spring force pulling ℓ_i down when the midpoints of ℓ_i and ℓ_j are at the same y-coordinate. This provides just the right amount

Algorithm 6.1: RepulsiveForce(ℓ_i, ℓ_j)

```
Input: labels $\ell_i$ and $\ell_j$
Output: repulsive force $r(i,j)$ that $\ell_j$ exerts upon $\ell_i$

$d \leftarrow \delta(i,j)$
if $d = 0$ then
    return $\sigma(i,j) \cdot F_{limit}$
else if $d \leq 1$ then
    $v \leftarrow \big(1/d^2 - \big(1 - |F_i^s|\big)\big)$
    if $v > F_{limit}$ then
        return $\sigma(i,j) \cdot F_{limit}$
    else
        return $\sigma(i,j) \cdot v$
else if $d \leq 2$ then
    return $\sigma(i,j) \cdot |F_i^s| \cdot \big(1 - (d-1)^2\big)$
return 0
```

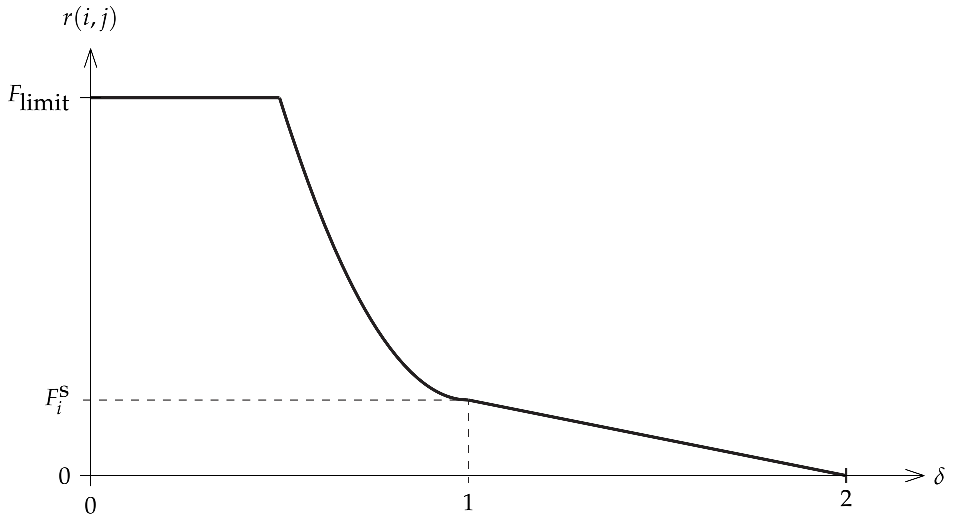


Figure 6.2: The graph visualizes the relation of the repulsive force r (y-axis) and the distance between two labels according to our distance function δ (x-axis).

of force to prevent overlaps while allowing the corresponding leaders to return to their default height if possible.

6.2.4 Total Force

Intuitively, the total force F_i acting on a single label ℓ_i consists of the sum of the aggregate repulsive forces F_i^{r} and the spring force F_i^{s}:

$$F_i := F_i^{\mathrm{r}} + F_i^{\mathrm{s}}.$$

This turned out to be too simplistic. What happens when the three labels ℓ_1, ℓ_2, and ℓ_3 are stacked? If ℓ_1 exerts the same force on ℓ_2 as ℓ_3 does, the forces cancel each other out. The remaining force, namely the spring force F_2^{s}, will probably cause an overlap. On that account, we separate positive forces F_i^+ and negative forces F_i^-. Formally,

$$\begin{aligned} F_i^+ &:= \textstyle\sum_{j \in I \setminus \{i\},\, r(i,j) > 0} r(i,j) \text{ and} \\ F_i^- &:= \textstyle\sum_{j \in I \setminus \{i\},\, r(i,j) < 0} r(i,j). \end{aligned}$$

Since $r(i,j) = 0$ does not have any effect, we omit this case in the above definitions. We finally include F_i^{s} in the total force if either $F_i^+ = 0$ or $F_i^- = 0$. In other words, if a label is surrounded by at least two labels, we ignore the spring force:

$$F_i := \begin{cases} F_i^+ + F_i^- + F_i^{\mathrm{s}} & \text{if } F_i^+ = 0 \text{ or } F_i^- = 0 \text{ and} \\ F_i^+ + F_i^- & \text{otherwise.} \end{cases}$$

6.2.5 Temperature

For each label ℓ_i, we define a temperature $T_i > 0$ that acts as a scaling factor of the total force F_i. When the billboard b_i is placed, we assign the constant default temperature T to its label ℓ_i. For each label ℓ_i, we track the total force F_i between two frames: if the sign of F_i changes (whereas we judge a change of $F_i = 0$ to $F_i \neq 0$ also as a change of the sign), we reset the temperature to a constant T_{base}. We distinguish the two temperatures T and T_{base} and require that $T > T_{\mathrm{base}}$ in order to faster solve overlaps of billboards that are added to the queue I of currently placed billboards. If the sign of F_i remains the same, we multiply its temperature T_i by a constant $T_{\mathrm{step}} > 1$ in order to increase T_i slightly. The idea is to increase the speed of moving of ℓ_i in the case that the label does not have reached a suitable position, that is, either ℓ_i is still overlapped or the corresponding leader does not have the desired height. On the other hand, we must prevent that the temperature becomes arbitrarily large. We just upperbound the temperature by a constant T_{limit}.

6.2.6 Leader-Height Change

The combination of the total force F_i and the temperature T_i induces either an extension or a contraction of the leader of label ℓ_i. In order to compute the final leader-height change Δ_i, we scale the combination by a constant factor χ, namely

$$\Delta_i := \chi \cdot T_i \cdot F_i, \tag{6.8}$$

and apply it to the corresponding leader in world space.

Due to rounding errors, it is unlikely that F_i will ever reach an equilibrium. In order to prevent that labels continuously move up and down, we only change the leader height if the absolute value of F_i is larger than a constant $F_{\min}$.

6.2.7 Running Time

The complexity for setting up any of the algorithms is linear in the size of the input. Nevertheless, more interesting for us is the complexity of the algorithms for a single frame. The auxiliary algorithms need at most $\mathcal{O}(n)$ time. (Recall that n is the maximum number of placed billboards in a frame.)

Further, in every frame, the force-directed algorithm processes the following steps for each billboard b_i in the queue I of visible billboards:

1. Computing the spring force F_i^{s} needs $\mathcal{O}(1)$ time (see Equation 6.1).
2. Computing the aggregate repulsive force F_i^{r} needs $\mathcal{O}(n)$ time (see Equation 6.2 and Algorithm 6.1).
3. Computing and applying the change Δ_i of the leader needs $\mathcal{O}(1)$ time (see Equation 6.8).

Consequently, we have a total asymptotic running time of $\mathcal{O}(n^2)$ for each frame. Since we can control the number n of processed billboards, we can influence the maximum computation time of the algorithm. This is especially useful for embedded applications, which typically only have low computational power. In practice, $n = 10$ seems to be sufficient for navigational devices as the devices usually have quite small displays. Nevertheless, in Section 6.3, we show that, even with a rather large n, we obtain frame rates of more than 420 FPS.

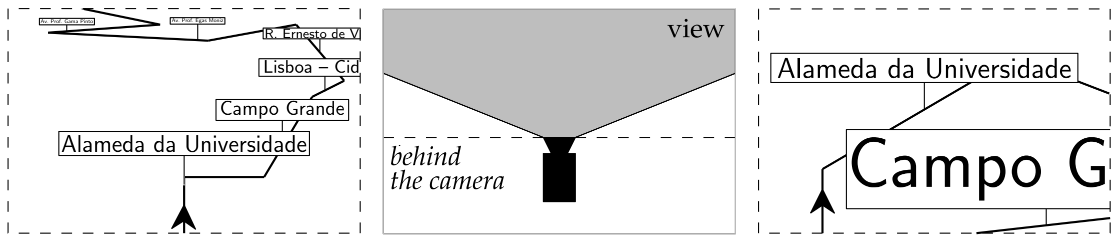
6.2.8 Implemented Improvements

In our above description of the force-directed algorithm, we constrained ourselves to the essence of the algorithm. Now, we present two additional modifications. The modifications do not influence the asymptotic running time.

Artificial Distances. According to our problem definition, labels must not overlap. Sometimes, it is yet desirable that two labels have a certain distance, a free space, between the their rectangles. To this end, we vertically enlarge the screen-space projection of each rectangle at its bottom by ν units, possibly pixels. Of course, this enlargement is transparent.

Relevant Billboards. The queue I of placed billboards is maintained by an auxiliary algorithm. There are, however, billboards for any given frame which are not necessarily relevant for the force-directed algorithm or even might cause undesired behavior. We want to ignore such billboards. For each billboard that is ignored, we reset all corresponding values to their defaults. Finally, in each iteration, we use $\hat{I} \subseteq I$ as the set of not-ignored, or *relevant*, billboards. We define n^* as the number of billboards in $\hat{I}$, or formally, $n^* := |\hat{I}|$.

Some billboards are so far away that we consider it unnecessary to include them in the computation of the forces; see Figure 6.3(a). For this reason, we ignore any billboard whose label has a screen-space area less than some value ξ. Such a billboard is still rendered but its label is not considered in force computations. Further recall that our billboards are placed in world space. For this reason, some billboards lie behind the camera; see Figure 6.3(b). This might cause problems when projecting them from world to screen space. In our force computations, we ignore any billboard that lies behind the camera. Additionally, we do not render it. Last, if a future part of the route is very near to the current position of the pointer, some labels become undesirably large; see Figure 6.3(c). Whenever the area of a label becomes larger than some value α, we ignore the corresponding billboard in force computations and do not render it.



(a) billboards in the far back are too small to read (b) billboards behind the camera might cause wrong projections (c) an adverse course of the route causes a very large label

Figure 6.3: Relevant and nonrelevant billboards (note: these are no screenshots).

6.3 Experiments

We have implemented our force-directed algorithm from Section 6.2, a testing environment, and a *static* algorithm for comparisons. For our implementations and tests, we used C++ with OpenSceneGraph 3.1[6.2] and Boost C++ Libraries 1.56[6.3] on Linux 3.16 with a 2.5-GHz Intel dual-core processor, 8 GB of RAM, and an Intel HD3000 integrated graphics card. We applied the GCC-4.9.2 compiler in 64-bit release mode. For the testing environment, we used an OpenStreetMap data set provided by Geofabrik[6.4] from which we extracted the downtown street network of Würzburg, a town of 120,000 inhabitants in Southern Germany. In order to simulate the navigation mode, we determined three different routes through Würzburg along each of which we created a camera path. We tested $n = 10$, 25, and 50. We think, however, that $n = 10$ is the most reasonable value for the rather small displays of navigational devices. Our virtual navigation system had a resolution of 1,366×768 pixels. In order to maintain n placed billboards for each frame and to guarantee paths of equal lengths, we stop the camera paths when the remaining part of the route has less than 50 reference points left. In total, the first, second, and third path consists of $N = 69$, 96, and 131 reference points, respectively. (Note that the total number of actually placed billboards is $N - 50 + n$ as we stop as soon as the remaining part of the route has less than 50 reference points left.) The paths need between 29 and 158 seconds. In our tests, the paths only pan and rotate the view. (We do not expect that the frame rate drops for the remaining interactions as there is no special handling for the different interaction types—neither in our algorithm nor for rendering.) In the camera paths for our tests, the pointer have a constant position and direction on the screen (as in Figure 6.3). As mentioned in the related-work section of Chapter 2, experts propose that billboards should shrink with distance to the user [MJD07b, VTW12]. We consider this advice in our implementation. Further we mentioned that experts suggest to seldom violating depth cues [MJD07a]. As we draw the billboards from back to front and we do not have any 3D objects, our depth cues are always correct.

For computing nice-looking, smoothly-moving labelings, we used the following empirical values. We used $h_0 = 5.0$ units in world space[6.5] as desired leader length. Further, we set the spring constant to $k = 0.25$ and the values of the temperatures to $(T, T_{\text{base}}, T_{\text{step}}, T_{\text{limit}}) = (1.0, 0.1, 1.05, 5.0)$. A force only manipulates the length of a leader if it is between the minimum-required force $F_{\min} = 0.1$ and the maximum-allowed force $F_{\text{limit}} = 5.0$. For scaling the aggregate repulsive force, we used $\rho = 1.0$; for scaling the change of the leader height, we used $\chi = 0.2$. We ignored labels with an area smaller than $\xi = 100$ pixels and labels that were larger than $\alpha = 0.25$ of the total resolution. The artificial distance between two rectangles should be $\nu = 5$ pixels. We removed a billboard as soon as its distance to the pointer became smaller than $\varepsilon = 10.0$ units in world space.

For the static algorithm, we ran the same camera paths as for the dynamic labeling algorithm using the same configurable values but we fixed the leader lengths to h_0.

Figure 6.4 shows some screenshots of the implementation of our algorithm. When we start the algorithm, the leader lengths are equal. In this example data set, the overlap

[6.2]`http://www.openscenegraph.org/`, accessed Nov. 28, 2014
[6.3]`http://www.boost.org/`, accessed Dec. 4, 2014
[6.4]`http://download.geofabrik.de/`, accessed Nov. 28, 2014
[6.5]For comparisons: we set the font size of the label text in world space to 3 units.

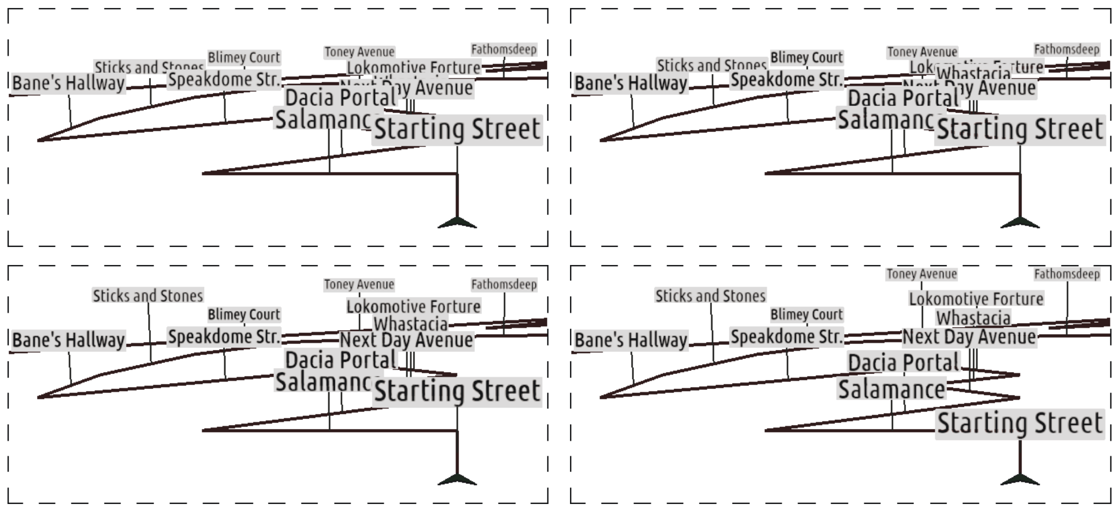


Figure 6.4: Screenshots of our program in an artificial data set. *In reading direction*: Within 1.5 seconds, the overlap of the initial label placement is resolved. On the very first subfigure, the total overlap is 6773 pixels.

resolves within two seconds. Figures 6.5 to 6.7 at the end of this chapter show some more screenshots of maps that we labeled with our algorithm. The amount of tilt we use for these pictures might be an unusual one for navigational devices. If we would, however, tilt the view less, we would see only a very small number of billboards. For this reason and as we set $n = 50$, at the back, billboards accumulate. (As the route in the figures leaves the view, the figures do not show all the 50 billboards.) While our algorithm obviously creates correct depth cues, sometimes the heights of the leaders seem unusual. See, for example, the billboards “Sieboldstraße-4” and “Sieboldstraße-5” in Figure 6.5. We might expect that the leader of “Sieboldstraße-5” is longer than the leader of “Sieboldstraße-4” as “Sieboldstraße-4” is, measured along the route, nearer to the pointer. It would be interesting to test if this disturbs the user. We finally remark that, for many streets, there is no name given in the input data. If we do not know the name of a street, we just use the name of the preceding or the succeeding street. This is why it seems as if we label the same street several times.

As mentioned at the beginning of this chapter, there is also a video that shows our algorithm in action[6.6]. The camera path used in the video was made for presentation; it is not a camera path we used for our tests.

In Table 6.2 and 6.3, we summarize our results for the force-directed algorithm and the static algorithm, both applied to the real-world data set. Table 6.2 first shows the number of placed billboards n, the number of relevant billboards n^*, and the number of visible relevant billboards n'. Note that the number of relevant billboards n^* influences the computation time of the force-directed algorithm. We decided to average the values over the number of visible relevant billboards n' as only these billboards are visible for

[6.6]`http://lamut.informatik.uni-wuerzburg.de/dynaroutelab.html`

billboards			static		force directed	
			frame rate	overlap	frame rate	overlap
n	n^*	n'	FPS	pixels	FPS	pixels
10	8	5	503	3,004	499	24
25	19	10	478	4,753	472	86
50	34	15	437	5,782	426	131

Table 6.2: Some results of our experiments for the static and the force-directed algorithm regarding the number of processed annotations, computation time, and overlapped area. The *frame rates* are averaged over the total computation time; the remaining values are averaged over frames. For each measurement, we assured that at least 50 streets were still ahead.

force directed: leader heights				
	leader height			deviation from h_0
n'	minimum	maximum	average	average
5	0.0	750	47	19
10	0.0	749	44	23
15	0.0	749	41	25

Table 6.3: Some values with regard to *leader heights* in the view computed by the force-directed approach. We give the values in pixels. The *minimum* and *maximum* leader height is absolute, the *average* values are averaged over frames and the number of visible relevant billboards n'.

the user and thus only these billboards can disturb the user. We further present the frame rates for each value of n, averaged over the three different paths. Our algorithm yields very good frame rates of more than 420 FPS when it places $n = 50$ billboards or less. If we only render the active route, the billboards, and the pointer (in other words, we do not render the street network), the frame rate increases by about 170 FPS. On the other hand, Table 6.2 shows that the frame rates of the static algorithm are only better by a few frames compared to the force-directed algorithm. This is due to the fact, that, for the static algorithm, we stretch the limits of what OpenSceneGraph combined with the integrated graphics card can achieve. To verify this, we also tested for *one* map the frame rate while OpenSceneGraph idled: without any computations or rendering but with loading the map and the route, we reach frame rates of about 450 FPS at our system.

Moreover, Table 6.2 shows the overlapped area. For each path and each value of n, we recorded the number of overlapped pixels of labels of visible relevant billboards. We divided the total number of overlapped pixels by the total number of frames. Table 6.2

shows that, compared to the static algorithm, our force-directed algorithm reduces the number of overlapped pixels per frame by about 98%.

In Table 6.3, we further present some values regarding the leader heights. We only consider relevant billboards that are also visible. The minimum and maximum leader height is an absolute value. We observe that it is common that labels are pressed downwards until they touch their reference points. On the other hand, the maximum leader height of less than 750 pixels shows that, in our tests, labels sometimes leave the view at the top (the height of our screen was 768 pixels; note that the reference points are spread over the view). For our tests we used the same, rather uncommon, amount of tilt as in Figures 6.5 to 6.7. For this reason, we expect that labels stay in the view with a more typical tilt. Nevertheless, in Section 6.4 we give an idea how to overcome this problem.

If we, however, consider the leader height averaged over all frames, each leader has a height of about 40 pixels; this corresponds to 1/20 of the height of the screen. This seems to be a very low leader height but we should not forget that billboards shrink with distance to the user. As some leaders are very long while others are very short, we also measured how much, averaged over frames, the actual height of the leaders differ from the desired height, namely about 20 pixels each. We also measured how much each label have to move such that it is overlap-free whereas we neglected emerging overlaps. As the overlapped area is already quite small, we observed that each label would have to move less than 0.3 pixels.

Unfortunately, we cannot compare our results regarding the computation time to the results of the algorithm introduced by Maass and Döllner [MD06]. They only state that their algorithm "operates in real time". Similarly, we cannot compare them to the results given by Gemsa et al. [GNN13] as they compute the entire labeling in advance, that is, the frame rate is determined by the rendering algorithm only but not by the labeling algorithm. Vaaraniemi et al. [VTW12] state that their algorithm has a frame rate of 180 FPS for computing label positions if they disable the rendering. If we only render the important part of our visualization, that is, the route, the labels, and the pointer, for $n = 50$, we obtain frame rates of almost 610 FPS. We conclude that our algorithm for attaching billboards to active streets in interactive maps with a perspective view in navigation mode yielding frame rates of more than 420 FPS is highly real-time capable and it is really worth trying to combine it with an algorithm that labels the remaining streets embedded.

6.4 Extensions

In this section, we give some ideas that might improve our algorithm with regard to aesthetics and usefulness. We describe, for example, how we can modify our algorithm such that it is able to deal with different leader directions as well as with different *anchor points*, that is, the point where the leader touches the label. Moreover, we discuss the difficulties when placing several billboards for the same street, explain how we can apply our algorithm to maps that are in map mode, and point out other little improvements. We only present concepts while omitting details.

Anchor Point and Leader Direction. So far, we only placed billboards whose vertical leaders hit their labels at the center of the labels' bottom edges. When presenting the preliminary user study, we saw that there are further (more or less suitable) possibilities of how we could connect labels and their reference points (see Figure 2.2 on page 22). More precisely, in each frame, we could allow for a set of the following possibilities: varying a) leader lengths, b) anchor points, and c) leader directions. For b) and c), we could either allow for arbitrary *manifestations* or for a fixed set of specific manifestations; in the latter case, we should animate the transitions. For example, we could permit rotating the leaders by $k \cdot 45°,\ k \in \mathbb{N}$, and selecting one of the four corners of the labels as anchor points.

First assume that we extend our model by arbitrarily varying anchor points (while maintaining that leaders change only vertically), that is, the lower edge of a label can slide along the touching point of its leader. To this end, we just adapt our algorithm by additionally considering forces that act horizontally. In order to guarantee that, while sliding horizontally, the leader keeps connected with its label, we should use another temperature that depends on the distance between the current anchor point and the corresponding label corner.

Now assume that we, additionally to the sliding of labels, permit a set of fixed leader directions. It should suffice to use the horizontal force added before and to vary the leader lengths according to the current direction. We should test if we can reduce the total force if we switch to another manifestation or if we rather should solve the current overlap by varying the length of the leaders and/or sliding the labels. In the case of a change, we should animate the transition. We also might include other criteria in our decision, for example, we could prefer a vertical leader to a horizontal one or sliding the label to varying the length of the leader.

For labels that move about freely, several force-directed algorithms for drawing graphs and also one for labeling interactive city maps [VTW12] have been introduced. Nevertheless, we think that too much motion overly attracts the users attention or the user might lose context, especially in maps that are in navigation mode.

Billboard of the Current Street. This is rather a subjective, user-dependent question. So far, we just remove the billboard of the street that we are currently driving along. Another possibly is to animate how the label embeds. If yet the street is very long, it could be desirable to know the name of the street for quite a long time. To this end, the pointer could push the billboard through the street, that is, we fix the reference point at the frontmost point of the pointer while the billboard is still placed at its reference point. The billboard vanishes when the user leaves the street.

Multiple Billboards. Hitherto, we attach one billboard to each active street. In general, for our algorithm, it does not matter how many annotations of the same street exist—in the case of multiple annotations per street, there is no need for any exception handling. Nevertheless, note that the distance between two reference points in world space strongly differs from the actual distance in screen space; in screen space, the distance in the front is larger than the distance in the background. Assuming that we always place each single billboard in the view, a too small distance certainly results in many label–label overlaps in the background. To this end, we recommend to refrain from placing a billboard between each two junctions of a street in order to obtain a unique label–object association for each point of the route. We expect that, in the case of using billboards, such an approach leads to an overloaded labeling and thus to a labeling that is unaesthetic and not very useful. Instead we propose to smoothly blend in new billboards if the unlabeled part of a street that is not too far away from the user is large enough. To keep user disturbances small, we suggest selecting the height of the leader such that the new billboards does not (or hardly) influence the current labeling. If the distance of two reference points of the same street becomes too small, we should blend out one of these billboards smoothly.

Maps in a 2D View. Our visualization framework automatically shrinks billboards with distance to the user if we place the billboards in world space. Rendering billboards placed in world space in a 2D view means that labels touch their reference points. Leaders still exist but they are not visible for the user; they are occluded by the labels. As leaders vary vertically in world space, in a 2D view, billboards only can change the distance to the user. The effect of the forces gets completely lost. If we place, however, the billboards in screen space, we can use leaders again. Although our algorithm is designed for 3D views, it is applicable for 2D views. We yet doubt that billboards in a 2D view are aesthetic.

Map Mode. Recall that our algorithm labels active streets in an interactive map in navigation mode whereas we assume that we only know the currently visible part of the route (indeed, we work with a slightly larger part; this is not necessary, though). Consequently, we can apply our algorithm in a map mode without any changes. If, however, the number of point features to be labeled is too large, there possibly is no space for moving labels. For that reason, we suggest labeling objects of a low frequency, for example, mountains of mountain chains, or reasonably selecting the point features to be labeled. Additionally, we recommend to place a billboards that left the view completely at a new position within the view if the corresponding object, for example, if it is an area feature, is still visible. In conclusion, for placing billboards in the map mode, it suffices to determine a proper set of reference points and to apply our algorithm.

A further extension could be that a label comes to front if the user clicks on it. When the user releases, depth cues are correct again.

Weights. As we already know, a billboard should shrink with distance to the user and should not consider the weight of its corresponding street [VTW12, MJD07b] (see the related-work section of Chapter 2). Moreover, we think that the billboard that the pointer reaches next is the most important one. On that account, we propose to indicate the type of a street by a style element like color or the font instead of indicating its weight by the height of the corresponding label. Nevertheless, our algorithm is able to deal with different label heights without any adaptions.

Considering the Map Background. We stated in the related-work section of Chapter 2 that the legibility of label text is almost unaffected if we use semi-transparent backgrounds [HV96] (as long as the transparency is not more than 50%). With this little workaround, we can make visible large parts of the background.

On the other hand, we could prefer that some elements are indeed not overlapped, for example, highway junctions. We easily can block areas that should be as overlap-free as possible by placing unmovable objects that repel labels. This increases the running time of our algorithm to $\mathcal{O}((n+o)^2)$, where o is the maximum number of blocking objects in a frame and under the assumption that the objects have a reasonable shape, for example, they are rectangles.

Eliminating Partly-Visible Labels. Labels that are only partly visible can be left as they are, forced into the view, or removed. So far, we leave partly-visible labels as they are. If we would remove them, we additionally have to avoid that billboards flicker. We could, for example, apply a waiting list as in Chapter 4. Finally, at the the top, left, and right edge of the view, we could place a bar that acts as *blocking* element forcing labels into the view. Labels that leave the view sideways, however, can only be avoided if labels can slide at their anchor points.

If a blocking bar is placed at the top edge, labels cannot leave at the top of the view. This, of course, causes further overlaps at the back. Note that, in most cases, billboards enter the view from the top of the view. This makes the back even more crowded. Nevertheless, as billboards in the background are not such important, we think that overlapping labels in the background are more desirable than labels leaving the view. If billboards come to the front when the user moves along the route, that is, when billboards get more important, overlaps solve automatically; thus, important labels are overlap free. Additionally, we could apply the same idea as for the map mode: if the user clicks on a label, it gets overlap-free.

6.5 Concluding Remarks

We have introduced a force-directed algorithm for DYNAROUTELAB, that is, for the problem of placing billboards with leaders of dynamically varying lengths to active streets in interactive maps with a perspective view. In our approach, each reference point tries to keep its corresponding leader at a desired length; overlapping labels repel each other. From frame to frame, we minimize the unbalanced forces. This yields labelings that avoid label–label overlaps of billboards that are near to the user but accepts overlaps in the background. Our algorithm directly reacts to changes of the current view by smoothly-moving overlapping labels. In our tests on real-world data with a realistic number of billboards, our implementation reached interactive frame rates of more than 420 FPS and reduced the number of overlapped pixels compared to the static algorithm (that does not avoid overlapping labels at all) by about 98%.

For the future, it would be interesting to improve our implementation such that it is able to support multiple billboards per street, different anchor points, and different leader directions. As we now have an implementation for placing embedded labels and placing billboards, it would also be interesting to verify the findings of our preliminary user study (in which we used static figures) by means of a user study that provides interactive scenarios. Other points to test could be if users prefer labels that are partly visible to labels that are not placed at all to billboards that leave the view at the top; or if the pointer should push a billboard through its street.

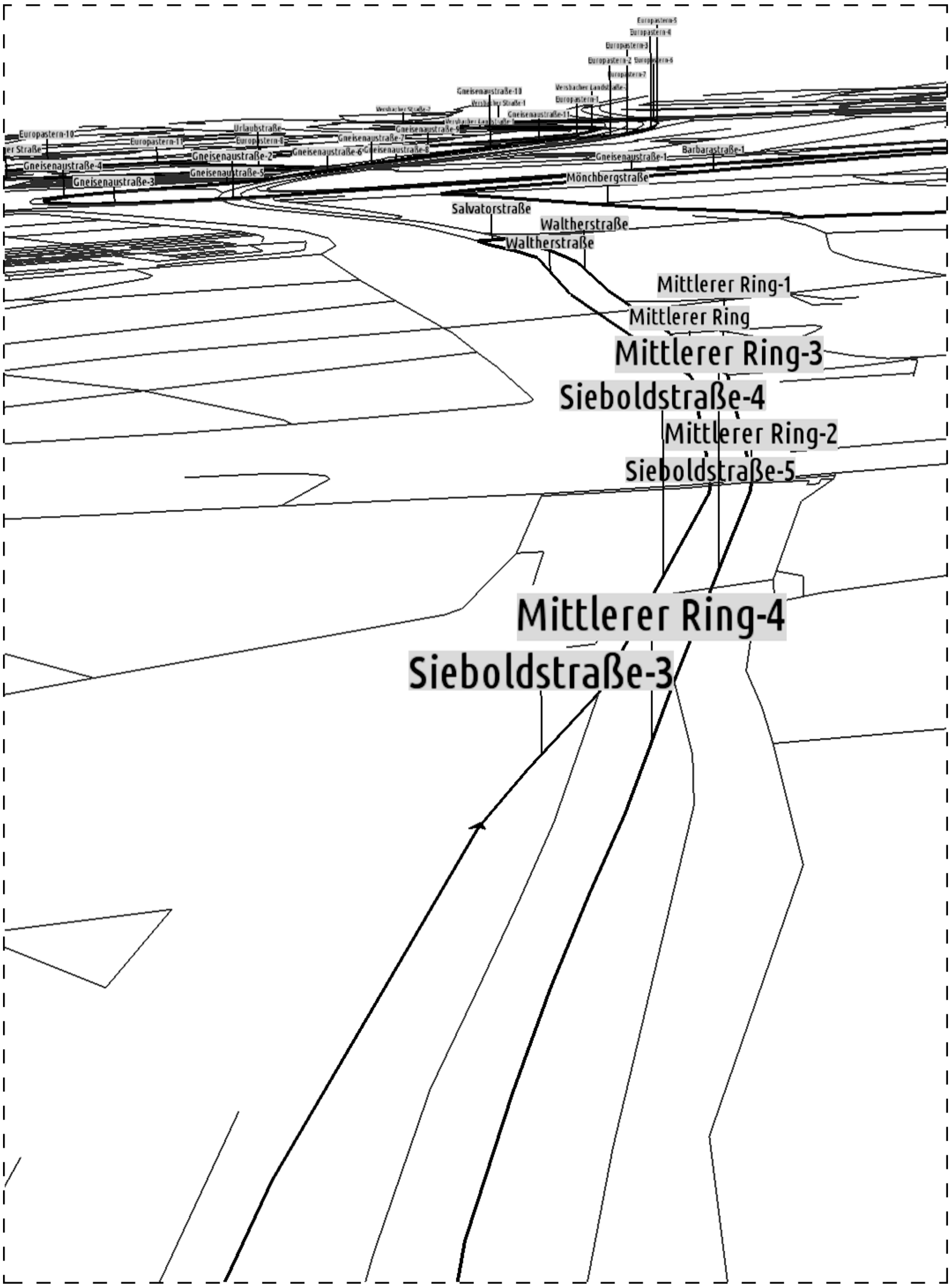


Figure 6.5: A map of Würzburg (Germany) labeled by our algorithm.

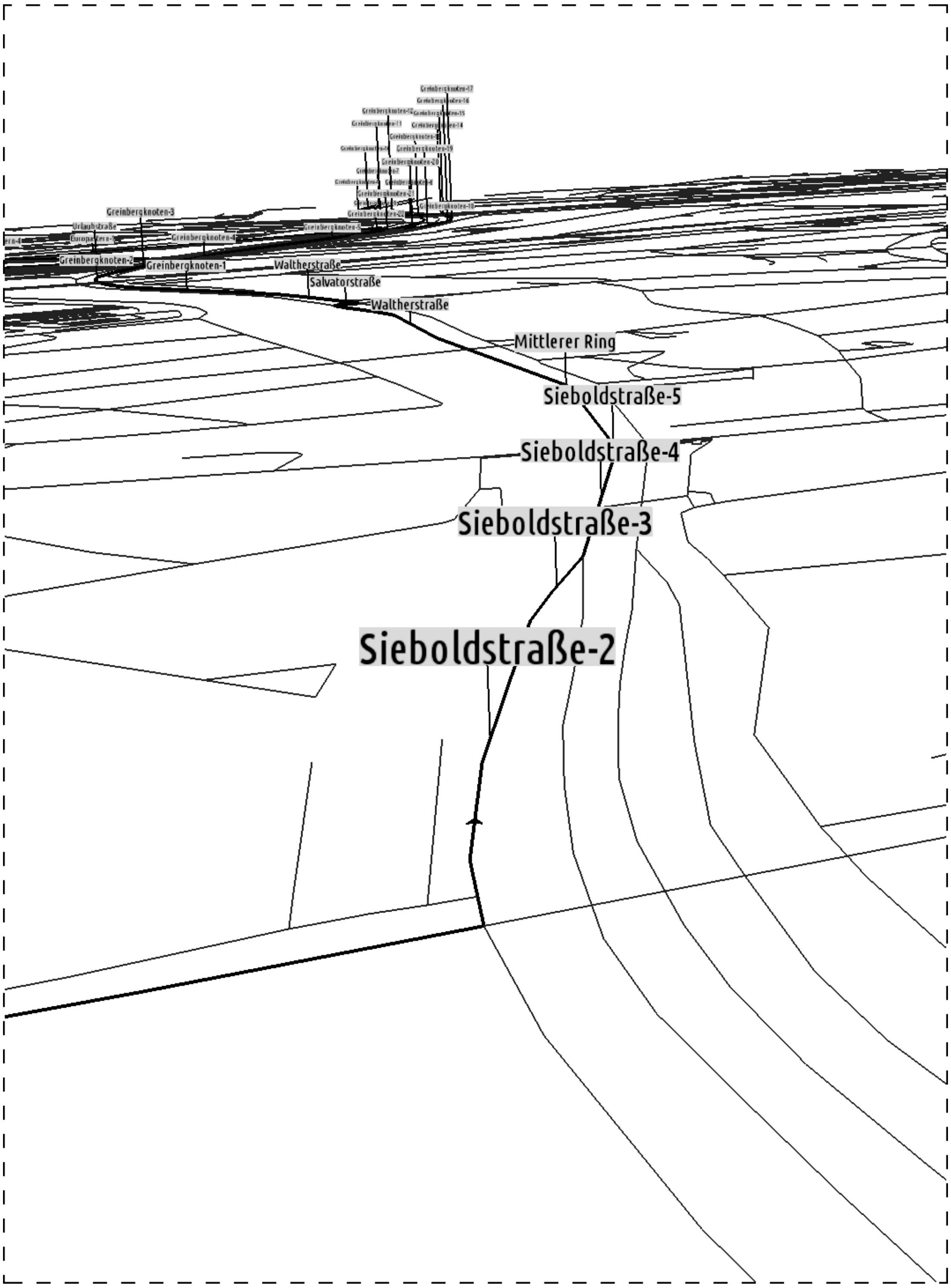


Figure 6.6: A map of Würzburg (Germany) labeled by our algorithm.

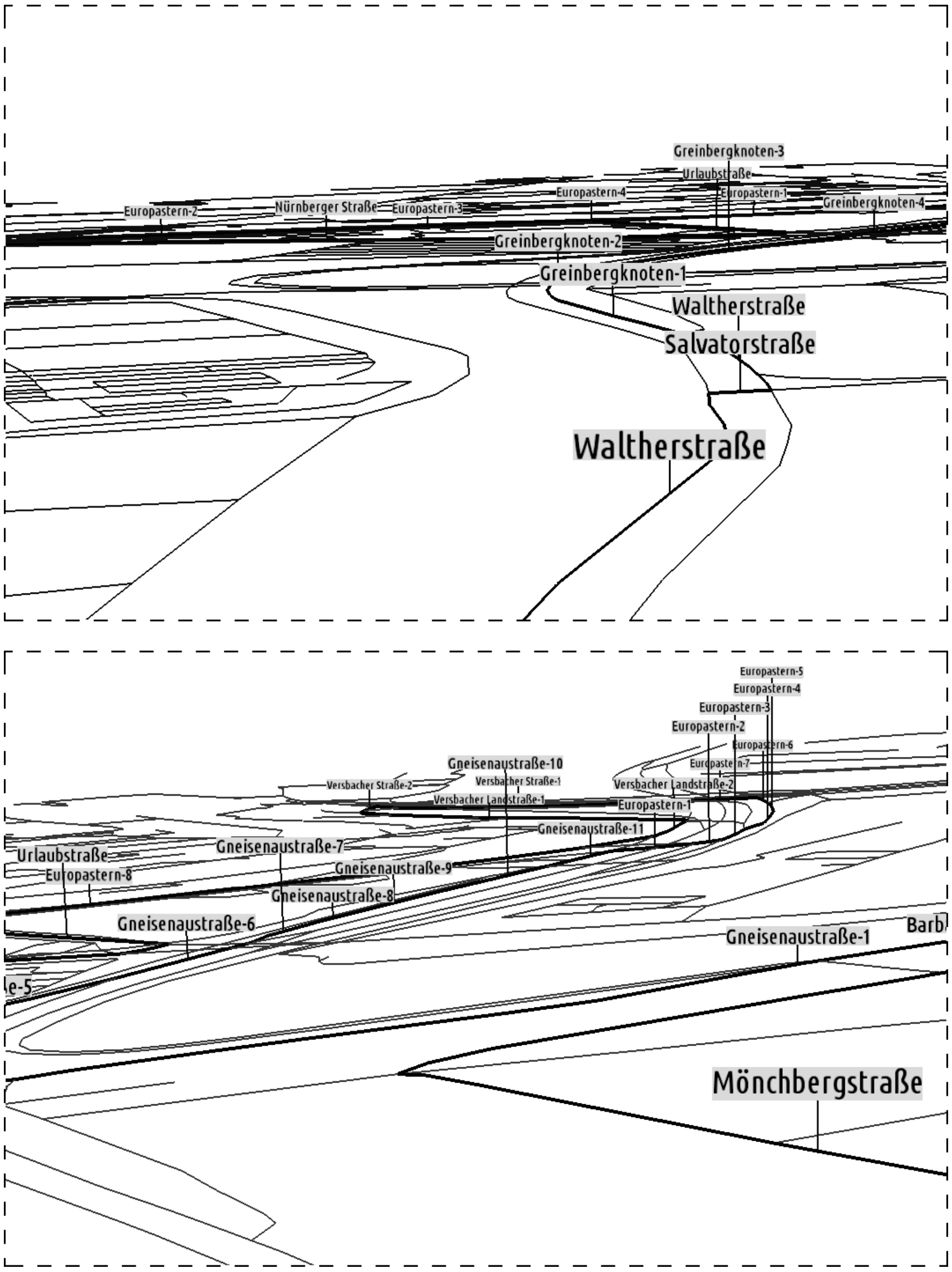


Figure 6.7: Two maps of Würzburg (Germany) labeled by our algorithm.

Chapter 7

Combination of Two Street-Labeling Algorithms: Embedded Street and Billboard Labeling

When using a digital map in navigation mode, labels along a user's route, that is, labels along active streets, are very important for the user. Consequently, these labels should be very well legible. On that account, we combine our algorithm for dynamically embedding labels into their streets and our algorithm for labeling active streets with billboards of dynamically varying leader lengths. We refer to the algorithm combining the two street-labeling algorithms as *combined algorithm* and the labeling using both labeling styles as *combined labeling*. Our aims for the combined labeling remain the same as for the single labelings: label as many different inactive streets as possible with embedded labels such that labels do not overlap at junctions and labels are aesthetically pleasing, keep the overlapped area between billboards as small as possible, and keep leaders as close as possible at a desired length. Yet, we do not prevent that billboards occlude embedded labels. In a dynamic scenario, it is unlikely that such occlusions persist for a long period of time. Thus, the implementation is straight forward.

Related Work In Chapters 5 and 6, we have already treated work which is also related to the combined labeling. On that account, we repeat it shortly.

Strijk [Str01] introduces an evaluation function which aims for finding label positions for embedded street labels in static maps such that the final labeling is aesthetic. In order to place many labels, he suggests splitting long street names into at most three parts and placing one of the parts above the street, one into the street, and the last one below the street. We use a similar approach: we also use an evaluation function but we use different evaluation criteria. We do not split street names. The most important difference is that our algorithm for embedded street labeling reacts to user interactions in real time, for example, if a label leaves the view completely, it is placed again within the view.

Maass and Döllner [MD07] present an algorithm to attach straight labels to streets in interactive 3D virtual environments. In order to detect label–label overlaps, they use a conflict graph. They provide two modes of user interactions. First, as soon as the user interacts, labels vanish; labels reappear when the user stops interacting. Alternatively, when the user interacts, labels stay constant on the screen; labels move to their new positions when the the user stops interacting. The second mode is similar to our approach: as soon as a label leaves the view completely, we place it again within the view (if possible). In our scenario, however, labels cannot be occluded by 3D objects.

In the related-work section of Chapter 6, we have given a table that compares our algorithm for placing billboards to similar approaches.

Gemsa et al. [GNN13] compute active ranges for labeling point features with rectangles while driving along a given route. Compared to our approach, Gemsa et al. do not use leaders, they cannot immediately react when the user takes another route, and their approach is for maps in a 2D view only.

Maass and Döllner [MD06] attach billboards with dynamically varying leader lengths to point features in interactive 3D virtual environments. In each frame, they compute the labeling from scratch using a greedy algorithm. Thus, other than our algorithm, their algorithm does not consider the history of the labeling. Consequently, labels can jump.

Finally, Vaaraniemi et al. [VTW12] pursue an idea that is similar to ours. They attach billboards with dynamically varying leader lengths to point features. In order to solve label–label overlaps, they use a force-directed approach where labels can move in any direction. In our case, labels move only vertically and their lowest positions are bounded by the corresponding reference points. Additionally, we only label the active route with billboards.

Our Contribution We have already introduced the concepts of the algorithms for labeling streets with embedded labels and with billboards in Chapter 5 and 6, respectively. Therefore, in this chapter, we only give the results of the implementation of the combined algorithm (see Section 7.1). We first explain the setup of our experiments; then, we discuss the resulting frame rates and consider factors that influence these frame rates. We go without any measurements of the labeling quality as we have evaluated the quality of our embedded street labeling and our billboard labeling in Chapter 5 and 6, respectively. We have made a video that shows the outcome of the combined algorithm[7.1].

To the best of our knowledge, hitherto, neither there is other scientific work that deals with annotating inactive streets by embedded labels and active streets by billboards nor there are navigational devices that already use such a labeling. On that account, we have submitted our idea as invention disclosure [NSW12].

7.1 Experiments

For these final tests, we used the algorithms for visualizing the street network of Section 5.3, the algorithm for embedded street labeling of Section 5.2, and the algorithm for

[7.1] `http://lamut.informatik.uni-wuerzburg.de/dynalinelab.html`

labeling the active route with billboards of Section 6.2. In order to obtain the combined algorithm, we had to adapt two things. First, as long as a street is labeled by a billboard, it should not be labeled by an embedded label. To this end, we simply removed the active streets from the set of streets whose labels are embedded. Second, the street networks of the tests for the embedded street labeling and for the billboard labeling differed. In the experiments for both algorithms, we used a map from Geofabrik[7.2] that we edited such that is shows a part of Würzburg. For the billboard labeling, we just used this map. For the embedded street labeling, we edited the map as described in Section 5.3.1. The final map provided 620 polylines; we have shown it in Figure 5.20 (see page 105). For the combined algorithm, we decided to use the same network as for the embedded street labeling because the algorithm for embedded street labeling needs more computational power. Moreover, for the tests of the combined algorithm, we used almost the same system configuration as for the tests of the embedded street labeling. We used a slightly more recent version of OpenSceneGraph[7.3], namely version 3.1 (instead of version 3.0). We repeat that, for our experiments for the algorithm for embedded street labeling, we used C++ on a Windows 7 system with a 3.3-GHz AMD triple-core processor, 8 GB of RAM, and a GeForce GTX 460 graphics card, applying the Microsoft Visual Studio 2010 Ultimate compiler in 32-bit release mode.

In order to simulate the navigation mode for the combined labeling, we determined three different routes through Würzburg along each of which we created a camera path. Our routing algorithm, however, is quite simple. With the help of a depth-first search, it searches a path from start to destination. Thus, in the smaller network for the combined algorithm, we could not find routes with more than 53 streets. We reduced the number of placed billboards to $n = 10, 20$, and 30 (instead of $n = 10, 25$, and 50). In order to maintain n placed billboards for each frame and to guarantee paths of equal lengths, we stop the camera paths as soon as the remaining part of the route has less than 30 streets left. Our routes for the combined labeling provide $N = 44, 45$, and 53 streets. The camera paths need between 22 and 51 seconds; they pan and rotate the view. As the combined algorithm is especially useful for maps in navigation mode, we go without tests in which we simulate the screen of a monitor (as we did for the embedded street labeling). We set the resolution of our simulated navigational device to 1024×600 pixels and the camera angle to $21°$. Averaged over all paths and frames, this leads to 5 visible billboards per frame. We show this configuration in Figure 7.1.

For an aesthetically-pleasing labeling with smoothly-moving labels, we almost used the same empirical values as presented in Chapters 5 and 6. Compared to the test of the billboard labeling in Chapter 6, we zoomed out. Thus, we set the desired leader length to $h_0 = 10$ units (in world space), the font size to 10 units, and the margin between the rectangles to $\nu = 10$ pixels. We also adjusted the font size of the embedded street labeling.

[7.2] `http://download.geofabrik.de/europe/germany/bayern/unterfranken.html`, accessed May 20, 2014

[7.3] `http://www.openscenegraph.org/`, accessed Nov. 28, 2014

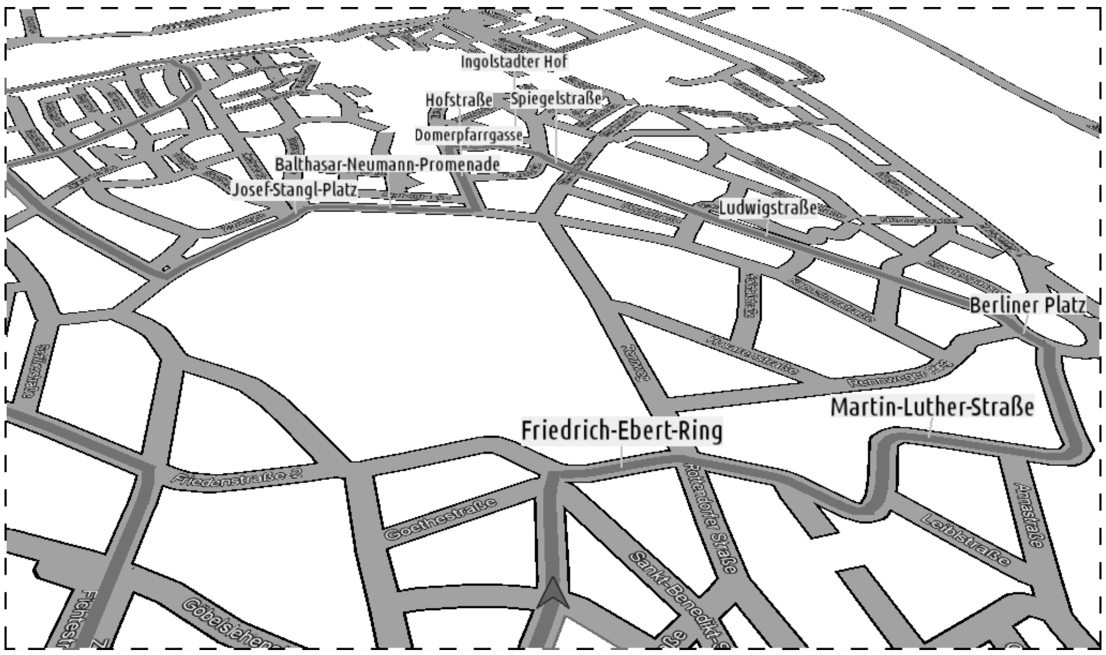


Figure 7.1: Screenshot that we took while processing one of our camera paths. The camera angle 21°, the size of the screen is 1024 × 600 pixels, the font size of the label text of billboards is 10 units (in world space).

We ran our combined algorithm on the, in total, nine camera paths described above. For comparisons, on each of these paths, we also ran our algorithm for embedded street labeling and our algorithm for placing billboards. We show the results of our tests in Table 7.1. Figures 7.2 to 7.5 at the end of this chapter show some screenshots of maps that were labeled by our combined algorithm. As mentioned above, a video showing our algorithm in action is available online[7.4]..

Table 7.1 shows that the number $n^* = 9, 16$, and 23 of relevant labels for the combined algorithm (averaged over the three paths and the total number of frames of each path) does not differ much from the number of relevant labels of the billboard labeling of Chapter 6 (which was $n^* = 8, 19$, and 34). The frame rate of the billboard labeling, however, decreases from more than 400 FPS to about 200 FPS. This is due to the fact that we used a different computer (particularly, for the combined labeling, we used Windows 7; for the billboard labeling, we used Linux 3.16). The number m of streets that are long enough to host an embedded label indicates how many streets (averaged over the paths and all frames) are tested either if a new label can be placed or if the current label still lies completely within the view. In our tests for attaching embedded labels to streets in a map in a 3D view (Chapter 5), in each frame, we had about 29 labels on the screen simulating a monitor; the average frame rate was 71 FPS. The frame rate of about 92 FPS for the combined labeling (averaged over all paths) is higher as, in the

[7.4] `http://lamut.informatik.uni-wuerzburg.de/dynalinelab.html`

computation time					
number of labels			∅frame rate [FPS]		
n	n^*	m	combined	embedded	billboard
10	9	55	94	96	218
20	16	55	91	93	208
30	23	55	92	94	211

Table 7.1: We present the *frame rate* for the *combined*, the *embedded*, and the *billboard* labeling in *frames per second*—depending on the number n of placed billboards and averaged over the three paths. We also measured the factors that influence the frame rate: the number n^* of relevant labels and the number m of streets that are long enough to host an embedded label; both depending on n, averaged over the three paths and over the total number of frames of each path.

camera paths for the combined labeling, we did not zoom. Table 7.1 also shows that the algorithm for placing billboards is very efficient: for $n = 10, 20$, and 30, compared to the embedded street labeling, we only lose 2 FPS each when labeling the map with both embedded labels and billboards. We finally conclude that the combined algorithm is still real-time capable.

7.2 Concluding Remarks

In Chapters 5 and 6, we have introduced algorithms for dynamically labeling inactive streets with embedded labels and for labeling active streets with billboards of dynamically varying leader lengths. For this chapter, we combined these two algorithms. Our aims for the combined labeling remained the same as for the two single labelings—in particular, we did not care about billboards occluding embedded labels. Moreover, we already have evaluated the quality of our labelings in Chapters 5 and 6. For these two reasons, in this chapter, we only presented results for the computation time and factors that influence the computation time. We reached frame rates of more than 90 FPS when having about 55 inactive streets (that are long enough to host their labels) in the view and processing about 20 labels with our force-directed algorithm. We conclude that the algorithm for simultaneously labeling inactive streets with embedded labels and active streets with billboards is still real-time capable for panning and rotation interactions.

For the future, it would be interesting to test whether users are comfortable with the combination of the two labeling styles and whether they benefit from it.

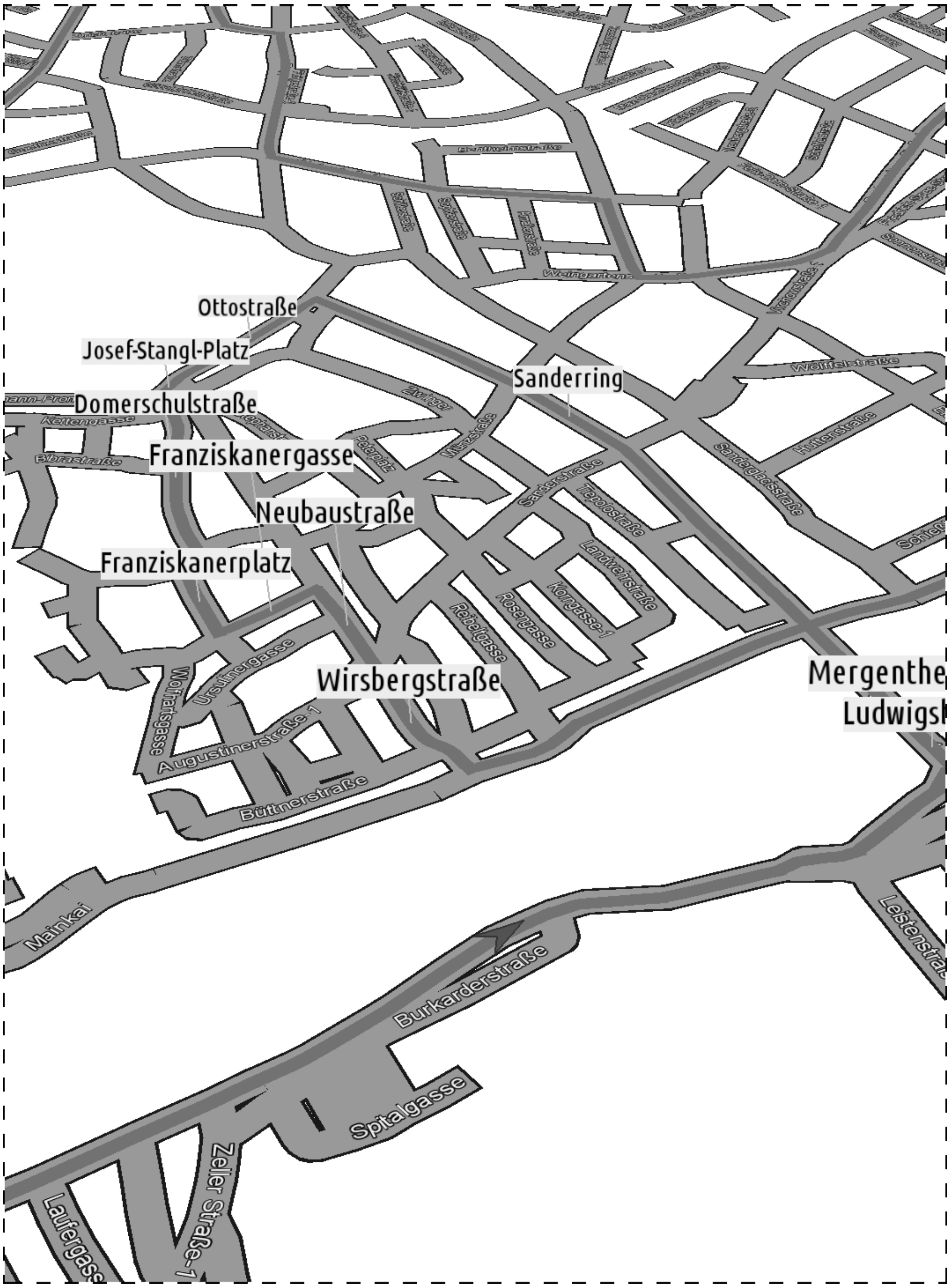


Figure 7.2: A map of Würzburg (Germany), computed by our algorithms.

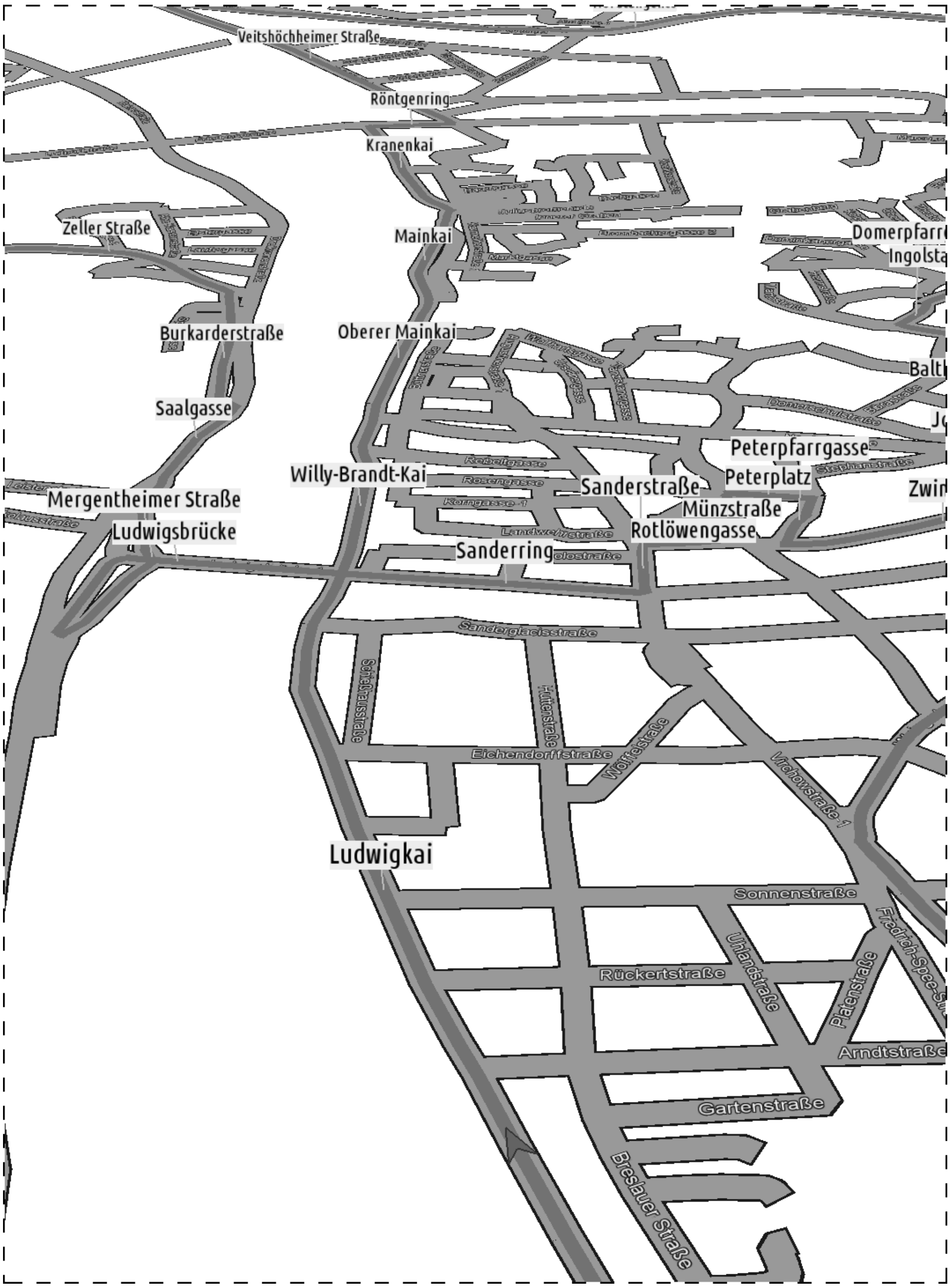


Figure 7.3: A map of Würzburg (Germany), computed by our algorithms.

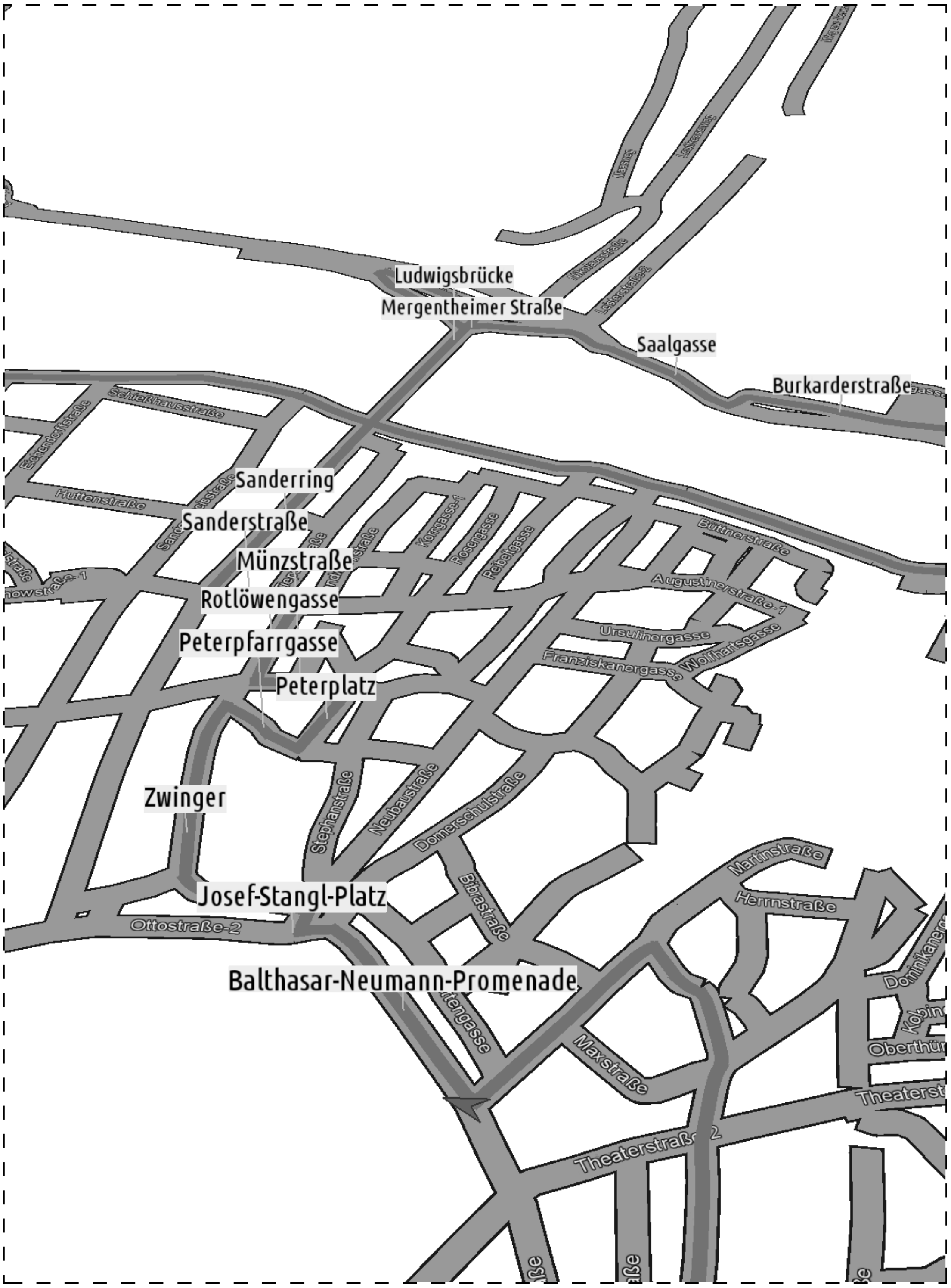


Figure 7.4: A map of Würzburg (Germany), computed by our algorithms.

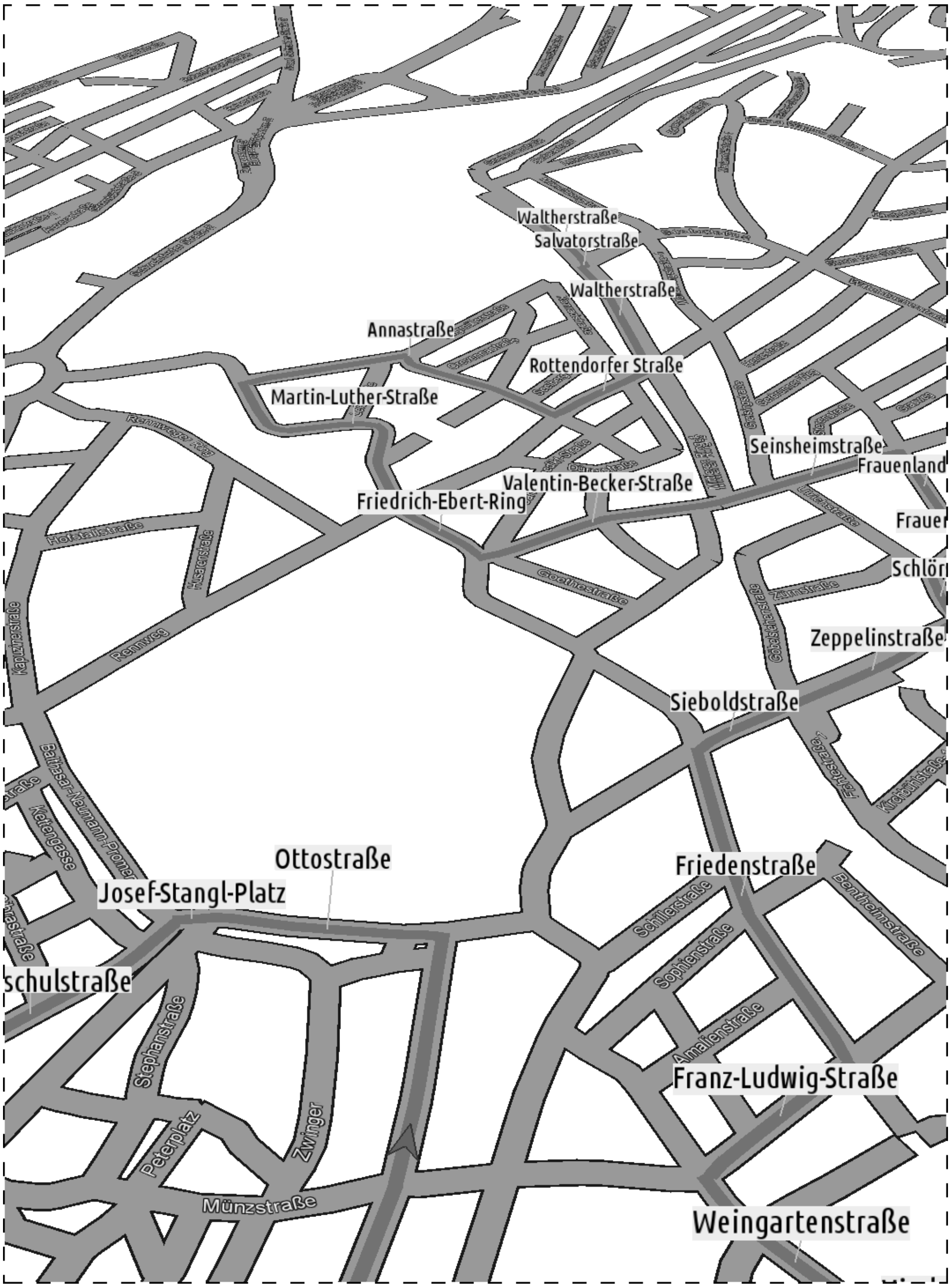


Figure 7.5: A map of Würzburg (Germany), computed by our algorithms.

Conclusion

In this thesis, we have introduced algorithms for dynamically labeling features in interactive maps. In general, our algorithms try to label as many different features as possible while prohibiting or at least trying to avoid label–label overlaps. Moreover, our algorithms consider the history of a labeling, that is, labels do not suddenly change their positions. As the related labeling problems for static maps are NP-hard, that is, we cannot even hope to efficiently compute an optimal solution for a single frame, we developed heuristics that compute nice-looking labelings in real time.

We have considered algorithms for labeling point and line features. We started with an algorithm that attaches labels to POIs. The algorithm computes a data structure of active ranges in a preprocessing and queries the current labeling at runtime. In our tests, the sum of the lengths of the active ranges computed by our heuristics was at least 85% of the same sum computed by a MIP with a gap of 35%. We went on with an algorithm that labels point features, for example, cities in small-scale maps, whereas we permitted each label to slide with its lower edge along its reference point. In order to save running time, we used a dynamic data structure, namely a rectangulation. Compared to a fixed-position model where the label's bottom edge is centered at the label's reference point, we improved the number of placed labels by 30–50%. In the second part of this thesis, we examined the problem of labeling streets in interactive maps. We first gave an algorithm that embeds labels into their features. For finding nice-looking label positions, we applied an evaluation function. We verified that the rather poor frame rates for zooming operations are a matter of the rendering routine. As embedded labels are sometimes hard to decipher but good legibility is essential for navigational devices, we complemented the algorithm embedding labels by an algorithm that attaches horizontally-written labels, that is, billboards, to streets along a user's route. In order to maintain an almost overlap-free labeling, we used a force-directed approach. Forces make overlapping labels repel each other and keep leaders as close as possible to a desired length. We experimentally showed that the overlap caused by our algorithm is less than 2% of the overlap caused by the algorithm with a fixed leader length.

For each of our algorithms, we provided a description of the concept, some extensions that we did not implement, and the results of tests on real-world data. We came to the conclusion that, in terms of computation time, most of our algorithms are already fast enough to be incorporated into a system for labeling digital maps, that is, they yield frame rates of more than 24 FPS; the algorithm placing embedded labels needs further improvements. Moreover, in contrast to our algorithms for labeling line features, our point-labeling algorithms do not support each interaction type, that is, they only provide a proper subset of the operations continuously panning, zooming, rotating, and tilting. Nevertheless, with our approaches, we cover a broad range of algorithms that are needed in a system for labeling digital maps. We dare to claim that we indeed made the gap between labeling static and interactive maps closer. There are, however, still several problems that should be tackled.

Concerning future work, it would be desirable to extend all of our algorithms such that each of them supports every interaction type. We have already stated that we doubt that our approach using active ranges (Chapter 3) can be adjusted such that we can label maps in a 3D view. On that account, new algorithms for attaching disk-shaped labels to point features in interactive maps should be designed. We also have investigated the problem of labeling point features with rectangular labels. Yet, the frame rate for zooming operations while annotating point features with sliding labels (Chapter 4) is quite low. It is not sure whether computation-time improvements, for example, using predictions as for panning operations, are indeed realizable. Moreover, it is not clear whether our ideas for labeling maps in a 3D view using a rectangulation are suitable. Possibly, completely new ideas are necessary.

Hitherto, we have not developed any special algorithm for labeling area features. An idea could be to stretch and contract horizontally-written labels such that the labels fit into the visible part of their corresponding area features. (Van Roessel [vR89] introduced an algorithm for annotating area features in static maps by horizontally-written labels.) On the other hand, in combination with dynamically labeling other features, this could cause too much movement. So, would it be reasonable to use static labels that simulate the shape of their area features (as proposed by Imhof [Imh75])? Kresse [Kre94] and Ropinski et al. [RPRH07] proposed such algorithms for maps in a 2D view and for 3D objects, respectively. The way we see it, labels of area features are less important in digital maps and even less important for navigational devices. On that account, it is acceptable that labels of area features are not updated if they are not visible any longer.

Another problem that should be tackled in a system for labeling digital maps is attaching pictorial labels to streets in maps that are in small scale, for example, placing highway signs. The challenge of this problem is to repeat the labels in a suitable way. On the one hand, labels should be repeated before they leave the view; on the other hand, especially at highway junctions, most of the streets should be labeled and the label–object associations must be obvious.

Finally, our algorithms should be adapted such that they react to obstacles at the map and they should use blending effects for labels that appear or vanish. Further, we should combine all of our algorithms in a single system. The subsequent step would be to adapt our algorithms such that they label objects in interactive 3D virtual environments while preventing occlusions by the 3D terrain or other 3D objects.

In this thesis, we have mainly considered label placement in practice. For each problem, we have implemented our algorithms and tested them on real-world data sets. We have measured the computation time and a value that indicates the quality of our labelings. We have compared these numbers to other, usually naive, approaches. Except for the problem of optimizing active ranges (Chapter 3), we did not compute optimal solutions or determine approximation factors. As our algorithms consider the labeling history, it is neither trivial to determine the approximation factors nor to develop algorithms that solve our problems optimally and run in acceptable time (computing optimal solutions for optimizing active ranges for point sets with 225 points needed more than three days of computation). For the future it would be very interesting to determine the factors and to verify the quality of our labelings by optimal solutions. Possibly, we should start with frame-wise comparisons.

Some of our decisions that have led to the final concepts of our algorithms are based on Imhof's rules [Imh75] for good label placement; other decisions are based on user studies that were conducted either by other researchers or by ourselves (Chapter 2); finally, several decisions are just our own subjective assessments. All of these rules and studies (but a few user studies by other researchers) are based on static pictures. It is not clear in how far the findings concerning aesthetics and usefulness differ for interactive maps, that is, for moving pictures. On that account it would be very interesting to conduct a conclusive user study in order to find out which of our concepts are accepted by and helpful for the users. Ideally, such a study should be realized in cooperation with psychologists.

Zusammenfassung

Gegenstand dieser Arbeit ist das Problem *interaktive Karten* zu beschriften und die dafür entwickelten Algorithmen auf ihre Praxistauglichkeit im Hinblick auf Rechenzeit zu testen. In der Informatik ist das *Beschriftungsproblem* ein Packungsproblem: Gegeben sei eine geometrische Form, ein sogenannter Behälter, sowie weitere geometrische Objekte. Lege so viele Objekte wie möglich in den Behälter, ohne dass sich zwei Objekte überlappen. Das Packungsproblem und das Beschriftungsproblem unterscheiden sich insofern, dass beim Beschriftungsproblem die möglichen Positionen jedes Objekts, oder vielmehr jeder *Beschriftung*, begrenzt sind. Daraus ergibt sich das allgemeine Beschriftungsproblem: Gegeben sei eine Menge von zu beschriftenden geometrischen Objekten (*Referenzobjekte*) in der Ebene und für jedes Referenzobjekt eine Menge von Beschriftungspositionen, sogenannte *Kandidaten*. Maximiere die Anzahl von gesetzten Beschriftungen, sodass jedes Referenzobjekt höchstens eine Beschriftung besitzt und keine zwei Beschriftungen überlappen. In Karten gibt es drei Arten von Referenzobjekten: Punkte, Linien und Gebiete. Leider können wir nicht davon ausgehen, dass es Algorithmen gibt, die das Beschriftungsproblem optimal und *effizient*, das heißt, mit kurzer Rechenzeit, lösen.

Interaktive Karten sind digitale Karten wie sie zum Beispiel in Navigationsgeräten verwendet werden. Interaktive Karten zeigen nur einen Ausschnitt der gesamten Karte, wobei der Benutzer diesen Ausschnitt, den *Sichtbereich*, verändern kann: Der Benutzer kann den Sichtbereich verschieben, verkleinern und vergrößern (das heißt, *heraus-* und *hineinzoomen*), ihn rotieren und die Ansicht *kippen*, also zwischen Draufsicht und Vogelperspektive variieren.

Diese spontanen Änderungen machen das Platzieren von Beschriftungen noch schwieriger. Sobald eine Beschriftung den Sichtbereich verlässt, sollte diese innerhalb des Sichtbereichs neu gesetzt werden. Beim Zoomen soll sich die Größe einer Beschriftung auf dem Bildschirm nicht ändern. Beim Herauszoomen müssen wir daher Beschriftungen löschen um Überlappungen zu verhindern. Beim Hineinzoomen entsteht Platz um weitere Beschriftungen zu platzieren. Diese Aktualisierungen müssen in *Echtzeit* durchgeführt werden, das heißt, sie müssen so schnell durchgeführt werden, dass der Benutzer nicht bemerkt, dass im Hintergrund neue Positionen berechnet werden. Eine weitere Anforderung interaktiver Karten ist, dass eine Beschriftung nicht *springen* oder *flackern* darf, das heißt, wenn eine Beschriftung ihre Position ändern muss, so soll sie sich kontinuierlich zu ihrer neuen Position bewegen, und, während der Benutzer hinauszoomt, darf eine gelöschte Beschriftung nicht wieder eingeblendet werden.

In dieser Dissertation stellen wir effiziente Algorithmen vor, die Punkte und Linien dynamisch beschriften. Wir versuchen stets so viele Referenzobjekte wie möglich zu beschriften, wobei wir gleichzeitig fordern, dass die platzierten Beschriftungen weder springen, flackern, noch sich überlappen. Wir haben unsere Algorithmen implementiert und mit Hilfe von echten Kartendaten getestet. Tatsächlich sind unsere Algorithmen echtzeitfähig.

Diese Dissertation besteht aus zwei Teilen. Im ersten Teil betrachten wir zwei Punktbeschriftungsprobleme. Zur Lösung des ersten Problems berechnen wir die Positionen der Beschriftungen bereits in einer Vorverarbeitung. Zur Laufzeit fragen wir diese nur noch ab. Wir konzentrieren uns auf den Fall, dass der Benutzer zoomt, das heißt, er verändert den *Maßstab* des Sichtbereichs. Während der Benutzer herauszoomt, werden die Abstände zwischen den zu beschriftenden Punkten auf dem Bildschirm kleiner. Da wir fordern, dass die Beschriftungen ihre Größe nicht ändern, müssen wir einige Beschriftungen entfernen, damit keine Überlappungen entstehen. Um zu verhindern, dass Beschriftungen springen, zentrieren wir sie auf ihren Referenzpunkten. Um zu verhindern, dass Beschriftungen flackern, weisen wir jeder Beschriftung ein zusammenhängendes Intervall von Maßstäben zu, in welchen sie letztendlich platziert werden. Wir stellen einen exakten Algorithmus vor, der auf ganzzahliger linearer Programmierung basiert, sowie Heuristiken, die auf einer Greedy-Strategie basieren. Da die Berechnung von optimalen Lösungen sehr zeitintensiv ist, verwenden wir den exakten Algorithmus nur um nachzuweisen, dass unsere Heuristiken in vielen Fällen fast-optimale Lösungen effizient berechnen.

Weiter vergleichen wir zwei *Modelle für Beschriftungen*, wobei die Beschriftungen zur Laufzeit berechnet werden. Wir repräsentieren eine Beschriftung durch ihr umschreibendes Rechteck. Zunächst beschreiben wir einen Algorithmus, der das *Schiebemodell* verwendet. Bei diesem kann sich ein Rechteck zur Laufzeit mit seiner Unterkante entlang seines Referenzpunktes bewegen. So schafft es Platz für weitere Rechtecke. Zudem beschreiben wir einen Algorithmus, der ein *Fest-Positionen-Modell* verwendet, bei welchem der Referenzpunkt stets in der Mitte der Unterkante des zugehörigen Rechtecks liegt. Wir kommen zu dem Schluss, dass die Anzahl der gesetzten Beschriftungen im Schiebemodell weitaus höher ist als die Anzahl der Beschriftungen im Fest-Positionen-Modell. Wir betrachten außerdem zwei verschiedene Methoden um zu testen, ob sich zwei Rechtecke überlappen. Einerseits betrachten wir einen naiven Ansatz, der jedes Paar von Beschriftungen auf Überlappung testet, andererseits nutzen wir eine einfache Datenstruktur um die Rechenzeit zu verbessern. Im Vergleich zum naiven Ansatz verbessert sich die Laufzeit unter Verwendung der Datenstruktur geringfügig.

Im zweiten Teil dieser Dissertation betrachten wir das Problem Straßen zu beschriften, wobei auch hier die Beschriftungen zur Laufzeit berechnet werden. Zunächst entwickeln wir einen Algorithmus, der Beschriftungen platziert, die dem Verlauf der zugehörigen Straßen folgen. Dabei achten wir darauf, dass Beschriftungen gut lesbar und optisch ansprechend sind. Dies erreichen wir beispielsweise, indem wir Beschriftungen setzen, die nur wenige Kurven haben. Unser Algorithmus bewertet jede einzelne Position entlang einer Straße hinsichtlich der Ästhetik der zu platzierenden Beschriftung und

entscheidet sich schließlich für eine Position.

Leider sind manche Beschriftungen, die dem Straßenverlauf folgen, nur schwer lesbar. Dabei ist es gerade für Navigationsgeräte besonders wichtig, dass die Beschriftungen entlang einer Route, die den Benutzer zu einem Ziel führt, gut lesbar sind. Aus diesem Grund erweitern wir den oben beschriebenen Algorithmus: Wenn die Ansicht der Karte in die Vogelperspektive gekippt ist und es eine Route gibt, platzieren wir entlang der Route achsenparallele Rechtecke. Wir setzen eine Beschriftung je Straße. Dafür bestimmen wir auf jeder Straße einen Referenzpunkt und verbinden die Beschriftung und den Referenzpunkt mit einem vertikalen Liniensegment, dessen Länge wir zur Laufzeit dynamisch variieren können. Um springende Beschriftungen zu vermeiden, erlauben wir, dass Beschriftungen überlappen, versuchen aber gleichzeitig die entsprechende Fläche möglichst klein zu halten. Weiter fordern wir, dass die Segmente möglichst eine bestimmte Länge haben. Wir stellen einen kräftebasierten Algorithmus vor, der die Längen der Liniensegmente beeinflusst: Rechtecke stoßen sich gegenseitig ab, Referenzpunkte ziehen ihre Rechtecke an oder stoßen sie ab. Verglichen mit einem Algorithmus, der die Längen der Segmente nicht verändern kann, reduziert unser Algorithmus die überlappte Fläche maßgeblich, wobei sich die Laufzeit kaum erhöht.

List of Publications, Invention Disclosures, and Internal Reports

Publications

[SAHW13] Nadine Schwartges, Dennis Allerkamp, Jan-Henrik Haunert, and Alexander Wolff. Optimizing active ranges for point selection in dynamic maps. In *Proceedings of the 16th ICA Generalisation Workshop (ICA'13)*. International Cartographic Association, 2013. 10 pages.

[Sch11] Nadine Schwartges. Approximationsalgorithmen für das gerichtete Maximum-Leaf-Spanning-Tree-Problem. Diploma thesis, Lehrstuhl I für Informatik, Universität Würzburg, March 2011. 59 pages.

[Sch14] Nadine Schwartges. Dynamic label placement in practice. In Stephan Mäs, Lars Bernard, and Hardy Pundt, editors, *Proceedings of the 2nd AGILE PhD School in 2013*, volume 1136 of *CEUR Online Proceedings*. CEUR-WS.org, 2014. 11 pages.

[SHWZ14] Nadine Schwartges, Jan-Henrik Haunert, Alexander Wolff, and Dennis Zwiebler. Point labeling with sliding labels in interactive maps. In Joaquín Huerta, Sven Schade, and Carlos Granell, editors, *Connecting a Digital Europe Through Location and Place (AGILE'14)*, Lecture Notes in Geoinformation and Cartography, pages 295–310. Springer-Verlag, 2014. Acceptance rate 32.8%.

[SMHW15] Nadine Schwartges, Benjamin Morgan, Jan-Henrik Haunert, and Alexander Wolff. Labeling streets along a route in interactive 3D maps using billboards. In Fernando Bação, Maribel Yasmina Santos, and Marco Painho, editors, *Geographic Information Science as an Enabler of Smarter Cities and Communities (AGILE'15)*, Lecture Notes in Geoinformation and Cartography, pages 269–287. Springer-Verlag, 2015. Acceptance rate 40.0%.

[SSW12] Nadine Schwartges, Joachim Spoerhase, and Alexander Wolff. Approximation algorithms for the maximum leaf spanning tree problem on acyclic digraphs. In Roberto Solis-Oba and Guiseppe Persiano, editors, *Proceedings of the 9th Workshop on Approximation and Online Algorithms (WAOA'11)*, volume 7164 of *Lecture Notes in Computer Science*, pages 77–88. Springer-Verlag, 2012. Acceptance rate 43.8%.

[SWH14] Nadine Schwartges, Alexander Wolff, and Jan-Henrik Haunert. Labeling streets in interactive maps using embedded labels. In Yan Huang, Markus Schneider, Michael Gertz, John Krumm, and Jagan Sankaranarayanan, editors, *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM-GIS'14)*, pages 517–520. ACM, 2014. Acceptance rate 44.0%.

Invention Disclosures

[NSW] Christian Neumann, Nadine Schwartges, and Alexander Wolff. *German version:* Verfahren und Vorrichtung zum Bestimmen einer Darstellung von Beschriftungselementen einer digitalen Karte sowie Verfahren und Vorrichtung zum Anzeigen einer digitalen Karte, 31 pages. *English version:* Method for labeling streets in navigation systems, 5 pages. 2012.

Internal Reports

[SHW11a] Nadine Schwartges, Jan-Henrik Haunert, and Alexander Wolff. Applicability of different map labeling methods, July 2011. 26 pages.

[SHW11b] Nadine Schwartges, Jan-Henrik Haunert, and Alexander Wolff. Formal definition of requirements for label placement in navigation systems, November 2011. 32 pages.

[SHW12a] Nadine Schwartges, Jan-Henrik Haunert, and Alexander Wolff. Embedded line labeling, September 2012. 11 pages.

[SHW12b] Nadine Schwartges, Jan-Henrik Haunert, and Alexander Wolff. Framework architecture and concept of a labeling algorithm, February 2012. 50 pages.

[SHW12c] Nadine Schwartges, Jan-Henrik Haunert, and Alexander Wolff. Interfaces of a new labeling algorithm, January 2012. 20 pages. Fully contained in [SHW12b].

[SHW12d] Nadine Schwartges, Jan-Henrik Haunert, and Alexander Wolff. Prototype: Labeling city features in dynamic maps at small scale, July 2012. 35 pages.

[SHW12e] Nadine Schwartges, Jan-Henrik Haunert, and Alexander Wolff. Prototype: Labeling city features in dynamic maps at small scale—3D interactions, December 2012. 8 pages.

[SS12] Leon Sering and Nadine Schwartges. Ausdünnung von Punktwolken, October 2012. 8 pages. German.

Acknowledgments

I would like to thank all people who made this thesis possible. First of all, I namely want to thank my supervisor Alexander Wolff. He offered me the opportunity to write this thesis and supported me when I visited conferences. We had many fruitful discussions. He was always sensitive and patient. When my employment contract with the University of Würzburg expired before I could finish my thesis, he found another opportunity for me to go on. In all the years, which I spent at his work group, I did not work for him but with him.

Second, I want to thank Jan-Henrik Haunert. I see him as my second supervisor. We also had many profitable discussions. He taught me the way to geo sciences. Even after moving from Würzburg to Osnabrück (where he was appointed professor), he supported me with helpful advice.

Further special thanks go to my research assistants Benjamin Morgan, Leon Sering, and Dennis Zwiebler (alphabetic order of surnames). They did not only implement large (or all) parts of my approaches but they also provided me with many good ideas that improved or enhanced my approaches. It was fun working with you!

I also want to thank the entire staff of the Chair of Computer Science I of the University of Würzburg and every other person in science who influenced my thesis in any way. I want to thank people with whom I discussed at conferences and those who awakened my interest in science. I want to thank the University of Würzburg for providing me with a grant. I want to thank the coordinator of the human-resource-development project of the University of Würzburg who guided my first steps from science to business.

Finally, I want to thank my family. First, I want to thank my partner who indulged my every whim. Although he did not study, he constantly discussed with me about my research and other problems. Many good ideas arouse. When I was writing day and night at this thesis, he cooked, brew tee, went grocery shopping, and fed my cat. In short, he ran the house. Thank you.

Last but not least, I want to thank the rest of my family. As far back as while studying, my family always had friendly words when I thought I cannot go on. True to the words

"Regardless of how dark the clouds above you are,
sooner or later, they will go away."

Bibliography

[Ali63] Georges Alinhac. *Cartographie Théorique et Technique*, chapter IV. Institut Géographique National, Paris, 1963. [pp. 4 and 15]

[AvKS98] Pankaj K. Agarwal, Marc van Kreveld, and Subhash Suri. Label placement by maximum independent set in rectangles. *Computational Geometry: Theory and Applications*, 11:209–218, 1998. [pp. 55 and 56]

[AW13] Anna Adamaszek and Andreas Wiese. Approximation schemes for maximum weight independent set of rectangles. In *Proceedings of the 54th Annual IEEE Symposium on Foundations of Computer Science (FOCS'13)*, pages 400–409. IEEE, 2013. [pp. 55 and 56]

[Bac05] Gerhard Bachfischer. Legibility and readability—a review of literature and research to understand issues referring to typography on screens and device displays. Technical Report IDWoP.tech.report.05.01, University of Technology Sydney, 2005. [pp. 17, 27, and 76]

[BDY06] Ken Been, Eli Daiches, and Chee Yap. Dynamic map labeling. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):773–780, 2006. [pp. 5, 34, 35, 54, 57, and 119]

[BM91] Kate Beard and William Mackaness. Generalization operations and supporting structures. In *Proceedings of the 10th International Symposium on Computer-Assisted Cartography (AutoCarto'10)*, pages 29–42. American Society for Photogrammetry & Remote Sensing, 1991. [pp. 4 and 32]

[BNPW10] Ken Been, Martin Nöllenburg, Sheung-Hung Poon, and Alexander Wolff. Optimizing active ranges for consistent dynamic map labeling. *Computational Geometry: Theory and Applications*, 43(3):312–328, 2010. [pp. 8, 31, 33, 34, 35, 37, 39, and 57]

[BSS07] Dirk Burghardt, Stefan Schmid, and Jantien Stoter. Investigations on cartographic constraint formalisation. In *Proceedings of the 10th ICA Workshop on Generalisation and Multiple Representation*. International Cartographic Association, 2007. [p. 32]

[CC09] Parinya Chalermsook and Julia Chuzhoy. Maximum independent set of rectangles. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'09)*, pages 892–901. Society for Industrial and Applied Mathematics, 2009. [pp. 55 and 56]

[CJ90] Anthony C. Cook and Christopher B. Jones. A Prolog interface to a cartographic database for name placement. In Kurt E. Brassel and Haruko Kishimoto, editors, *Proceedings of the 4th International Symposium on Spatial Data Handling (SDH'90)*, pages 701–710. The Ohio State University, 1990. [p. 4]

[dBCvKO08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*, chapter 2 and 6. Springer-Verlag, 3rd edition, 2008. [pp. 39, 54, and 58]

[dBG12] Mark de Berg and Dirk H. P. Gerrits. Approximation algorithms for free-label maximization. *Computational Geometry: Theory and Applications*, 45(4):153–168, 2012. [p. 5]

[Ead84] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984. [p. 121]

[ECMS97] Shawn Edmondson, Jon Christensen, Joe Marks, and Stuart Shieber. A general cartographic labeling algorithm. *Cartographica*, 33(4):13–23, 1997. [p. 86]

[EHJ$^+$10] Thomas Erlebach, Torben Hagerup, Klaus Jansen, Moritz Minzlaff, and Alexander Wolff. Trimming of graphs, with application to point labeling. *Theory of Computing Systems*, 47(3):613–636, 2010. [p. 56]

[EJS05] Thomas Erlebach, Klaus Jansen, and Eike Seidel. Polynomial-time approximation schemes for geometric intersection graphs. *SIAM Journal of Computing*, 34(6):1302–1323, 2005. [pp. 55 and 56]

[EKW05] Dietmar Ebner, Gunnar W. Klau, and René Weiskircher. Label number maximization in the slider model. In János Pach, editor, *Proceedings of the 12th International Symposium on Graph Drawing (GD'04)*, volume 3383 of *Lecture Notes in Computer Science*, pages 144–154. Springer-Verlag, 2005. [p. 122]

[FHS$^+$12] Martin Fink, Jan-Henrik Haunert, André Schulz, Joachim Spoerhase, and Alexander Wolff. Algorithms for labeling focus regions. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2583–2592, 2012. [p. 48]

[FPT81] Robert J. Fowler, Michael S. Paterson, and Steven L. Tanimoto. Optimal packing and covering in the plane are NP-complete. *Information Processing Letters*, 12(3):133–137, 1981. [pp. 4, 10, 53, and 55]

[FW91] Michael Formann and Frank Wagner. A packing problem with applications to lettering of maps. In *Proceedings of the 7th Annual ACM Symposium on Computational Geometry (SoCG'91)*, pages 281–288. ACM, 1991. [p. 5]

[Ger13] Dirk H. P. Gerrits. *Pushing and Pulling*. PhD thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, August 2013. [p. 36]

[GGD07] Rafal Goralski, Christopher M. Gold, and Maciej Dakowicz. Application of the kinetic Voronoi diagram to the real-time navigation of marine vessels. In *Proceedings of the 6th Computer Information Systems and Industrial Management Applications (CISIM'07)*, pages 129–134. IEEE, 2007. [p. 56]

[GHN11] Andreas Gemsa, Jan-Henrik Haunert, and Martin Nöllenburg. Boundary-labeling algorithms for panorama images. In Isabel Cruz, Divyakant Agrawal, Christian S. Jensen, Eyal Ofek, and Egemen Tanin, editors, *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM-GIS'11)*, pages 289–298. ACM, 2011. [p. 123]

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*, volume 29. W. H. Freeman and Co., New York, 1979. [p. 4]

[GNN13] Andreas Gemsa, Benjamin Niedermann, and Martin Nöllenburg. Trajectory-based dynamic map labeling. In Leizhen Cai, Siu-Wing Cheng, and Tak Wah Lam, editors, *Proceedings of the 24th International Symposium on Algorithms and Computation (ISAAC'13)*, volume 8283 of *Lecture Notes in Computer Science*, pages 413–423. Springer-Verlag, 2013. [pp. 36, 57, 122, 123, 135, and 144]

[GNN14] Andreas Gemsa, Benjamin Niedermann, and Martin Nöllenburg. Label placement in road maps. In *Proceedings of the 30th European Workshop on Computational Geometry (EuroCG'14)*, 2014. [p. 86]

[GNR11a] Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter. Consistent labeling of rotating maps. In Frank Dehne, John Iacono, and Jörg-Rüdiger Sack, editors, *Proceedings of the 12th International Symposium on Algorithms and Data Structures (WADS'11)*, volume 6844 of *Lecture Notes in Computer Science*, pages 451–462. Springer-Verlag, 2011. [pp. 35 and 57]

[GNR11b] Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter. Sliding labels for dynamic point labeling. In *Proceedings of the 23th Canadian Conference on Computational Geometry (CCCG'11)*, pages 205–210. Citeseer, 2011. [pp. 35 and 57]

[GNR14] Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter. Evaluation of labeling strategies for rotating maps. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA'14)*, volume 8504 of *Lecture Notes in Computer Science*, pages 235–246. Springer-Verlag, 2014. [p. 35]

[HH14] Jan-Henrik Haunert and Tobias Hermes. Labeling circular focus regions based on a tractable case of maximum weight independent set of rectangles. In Falko Schmid, Christian Kray, and Holger Fritze, editors, *Proceedings of the the 2nd ACM SIGSPATIAL Workshop on Interacting with Maps (MapInteract'14)*, 2014. Accepted for publication. [p. 48]

[Hir82] Stephen A. Hirsch. An algorithm for automatic name placement around point data. *The American Cartographer*, 9(1):5–17, 1982. [p. 122]

[HSKL05] Lars Harrie, Hanna Stigmar, Tommi Koivula, and Lassi Lehto. An algorithm for icon labelling on a real-time map. In Peter F. Fisher, editor, *Proceedings of the 11th International Symposium on Spatial Data Handling (SDH'05)*, pages 493–507. Springer-Verlag, 2005. [p. 56]

[HV96] Beverly L. Harrison and Kim J. Vicente. An experimental evaluation of transparent menu usage. In Michael J. Tauber, Victoria Bellotti, Robin Jeffries, Jock D. Mackinlay, and Jakob Nielsen, editors, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'96)*, pages 391–398. ACM, 1996. [pp. 16, 83, 121, and 138]

[HW10] Jan-Henrik Haunert and Alexander Wolff. Area aggregation in map generalisation by mixed-integer programming. *International Journal of Geographical Information Science*, 24(12):1871–1897, 2010. [p. 33]

[Imh75] Eduard Imhof. Positioning names on maps. *The American Cartographer*, 2(2):128–144, 1975. [pp. 4, 15, 16, 86, 112, 121, 154, and 155]

[JSI$^+$10] Jacek Jankowski, Krystian Samp, Izabela Irzynska, Marek Jozwowicz, and Stefan Decker. Integrating text with video and 3D graphics: The effects of text drawing styles on text readability. In Elizabeth Mynatt, Geraldine Fitzpatrick, Scott Hudson, Keith Edwards, and Tom Rodden, editors, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'10)*, pages 1321–1330. ACM, 2010. [p. 16]

[KGV83] Edward Scott Kirkpatrick, Charles Daniel Gelatt, Jr., and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983. [p. 122]

[KN85] Asher Koriat and Joel Norman. Reading rotated words. *Journal of Experimental Psychology: Human Perception and Performance*, 11(4):490–508, 1985. [pp. 15, 83, and 119]

[KN90] Jan Kratochvíl and Jaroslav Nešetřil. Independent set and clique problems in intersection-defined classes of graphs. *Commentationes Mathematicae Universitatis Carolinae*, 31(1):85–93, 1990. [p. 85]

[Kre94] Wolfgang Kresse. *Plazierung von Schrift in Karten*. PhD thesis, Faculty of Agriculture, University of Bonn, May 1994. [pp. 87 and 154]

[KS13] Janne Kovanen and L. Tiina Sarjakoski. Sequential displacement and grouping of point symbols in a mobile context. *Journal of Location Based Services*, 7(2):79–97, 2013. [p. 36]

[LSC08] Martin Luboschik, Heidrun Schumann, and Hilko Cords. Particle-based labeling: Fast point-feature labeling without obscuring other visual features. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1237–1244, 2008. [p. 57]

[LvDCR00] Kevin Larson, Maarten van Dantzich, Mary Czerwinski, and George Robertson. Text in 3D: Some legibility results. In James Begole, editor, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'00)*, pages 145–146. ACM, 2000. [p. 16]

[MD06] Stefan Maass and Jürgen Döllner. Efficient view management for dynamic annotation placement in virtual landscapes. In Andreas Butz, Brian Fischer, Antonio Krüger, and Patrick Oliver, editors, *Proceedings of the 6th International Symposium on Smart Graphics (SG'06)*, volume 4073 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 2006. [pp. 57, 122, 123, 135, and 144]

[MD07] Stefan Maass and Jürgen Döllner. Embedded labels for line features in interactive 3D virtual environments. In Hannah Slay, Stephen N. Spencer, and Shaun Bangay, editors, *Proceedings of the 5th International Conference on Computer Graphics, Virtual Reality, Visualization and Interaction in Africa (AFRIGRAPH'07)*, pages 53–59. ACM, 2007. [pp. 87, 110, and 143]

[MJD07a] Stefan Maass, Markus Jobst, and Jürgen Döllner. Depth cue of occlusion information as criterion for the quality of annotation placement in perspective views. In Monica Wachowicz and Lars Bodum, editors, *Proceedings of the 10th AGILE International Conference on Geographic Information Science (AGILE'07)*, Lecture Notes in Geoinformation and Cartography, pages 473–486. Springer-Verlag, 2007. [pp. 17 and 132]

[MJD07b] Stefan Maass, Markus Jobst, and Jürgen Döllner. Use of depth cues for the annotation of 3D geo-virtual environments. In *Proceedings of the 23rd International Cartographic Conference (ICC'07)*. International Cartographic Association, 2007. [pp. 16, 17, 18, 20, 21, 26, 73, 98, 132, and 137]

[Mot07] Kevin D. Mote. Fast point-feature label placement for dynamic visualizations. *Information Visualization*, 6(4):249–260, 2007. [p. 57]

[MS91] Joe Marks and Stuart Shieber. The computational complexity of cartographic label placement. Technical Report TR-05-91, Center for Research in Computing Technology, Harvard University, 1991. [p. 4]

[NPS10] Martin Nöllenburg, Valentin Polishchuk, and Mikko Sysikaski. Dynamic one-sided boundary labeling. In Divyakant Agrawal, Pusheng Zhang, Amr El Abbadi, and Mohamed Mokbel, editors, *Proceedings of the 18th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM-GIS'10)*, pages 310–319. ACM, 2010. [p. 36]

[NSW12] Christian Neumann, Nadine Schwartges, and Alexander Wolff. Verfahren und Vorrichtung zum Bestimmen einer Darstellung von Beschriftungselementen einer digitalen Karte sowie Verfahren und Vorrichtung zum Anzeigen einer digitalen Karte. Invention disclosure, July 2012. Submitted, 31 pages. [pp. 13 and 144]

[ODF09] Kristien Ooms, Philippe De Maeyer, and Veerle Fack. A user centered approach for dynamic map labeling. In Tijs Neutens and Philippe De Maeyer, editors, *Proceedings of the 15th International Conference InterCarto-InterGIS*, pages 173–180, 2009. [p. 16]

[PS05] Sheung-Hung Poon and Chan-Su Shin. Adaptive zooming in point set labeling. In M. Liśkiewicz and R. Reischuk, editors, *Proceedings of the 15th Symposium on Fundamentals of Computation Theory (FCT'05)*, volume 3623 of *Lecture Notes in Computer Science*, pages 233–244. Springer-Verlag, 2005. [p. 33]

[PSS$^+$03] Sheung-Hung Poon, Chan-Su Shin, Tycho Strijk, Takeaki Uno, and Alexander Wolff. Labeling points with weights. *Algorithmica*, 38(2):341–362, 2003. [pp. 4, 53, and 56]

[PT46] Donald G. Paterson and Miles A. Tinker. Readability of newspaper headlines printed in capitals and in lower case. *Journal of Applied Psychology*, 30(2):161–168, 1946. [p. 17]

[Reh81] Rolf F. Rehe. *Typographie: Wege zur besseren Lesbarkeit*. Verlag Coating Thomas & Co., St. Gallen, Switzerland, 1981. [p. 17]

[RPRH07] Timo Ropinski, Jörg-Stefan Praßni, Jan Roters, and Klaus Hinrichs. Internal labels as shape cues for medical illustration. In Hendrik P. A. Lensch, Bodo Rosenhahn, Hans-Peter Seidel, Philipp Slusallek, and Joachim Weickert, editors, *Proceedings of the Vision, Modeling, and Visualization Conference (VMV'07)*, volume 7, pages 203–212. Citeseer, 2007. [p. 154]

[SAHW13] Nadine Schwartges, Dennis Allerkamp, Jan-Henrik Haunert, and Alexander Wolff. Optimizing active ranges for point selection in dynamic maps.

In *Proceedings of the 16th ICA Generalisation Workshop (ICA'13)*. International Cartographic Association, 2013. 10 pages. [p. 9]

[SHWZ14] Nadine Schwartges, Jan-Henrik Haunert, Alexander Wolff, and Dennis Zwiebler. Point labeling with sliding labels in interactive maps. In Joaquín Huerta, Sven Schade, and Carlos Granell, editors, *Connecting a Digital Europe Through Location and Place (AGILE'14)*, Lecture Notes in Geoinformation and Cartography, pages 295–310. Springer-Verlag, 2014. [p. 10]

[SLG$^+$13] Anish Das Sarma, Hongrae Lee, Hector Gonzalez, Jayant Madhavan, and Alon Halevy. Consistent thinning of large geographical data for map visualization. *ACM Transactions on Database Systems (TODS'13)*, 38(4):22.1–22.36, 2013. [p. 37]

[SMHW15] Nadine Schwartges, Benjamin Morgan, Jan-Henrik Haunert, and Alexander Wolff. Labeling streets along a route in interactive 3D maps using billboards. In Fernando Bação, Maribel Yasmina Santos, and Marco Painho, editors, *Geographic Information Science as an Enabler of Smarter Cities and Communities (AGILE'15)*, Lecture Notes in Geoinformation and Cartography, pages 269–287. Springer-Verlag, 2015. [pp. 8 and 12]

[Str01] Tycho Strijk. *Geometric Algorithms for Cartographic Label Placement*. PhD thesis, Department of Computer Science, Utrecht University, January 2001. [pp. 15, 87, and 143]

[SU00] Sebastian Seibert and Walter Unger. The hardness of placing street names in a Manhattan type map. In Giancarlo Bongiovanni, Giorgio Gambosi, and Rosella Petreschi, editors, *Proceedings of the 4th Italian Conference on Algorithms and Complexity (CIAC'00)*, volume 1767 of *Lecture Notes in Computer Science*, pages 102–112. Springer-Verlag, 2000. [p. 85]

[SWH14] Nadine Schwartges, Alexander Wolff, and Jan-Henrik Haunert. Labeling streets in interactive maps using embedded labels. In Yan Huang, Markus Schneider, Michael Gertz, John Krumm, and Jagan Sankaranarayanan, editors, *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM-GIS'14)*, pages 517–520. ACM, 2014. [p. 12]

[Tin72] Miles A. Tinker. Effects of angular alignment upon readability of print. *Journal of Educational Psychology*, 47(6):358–363, 1972. [pp. 15, 83, and 119]

[vKSW99] Marc van Kreveld, Tycho Strijk, and Alexander Wolff. Point labeling with sliding labels. *Computational Geometry: Theory and Applications*, 13:21–47, 1999. [pp. 4, 53, and 56]

[vO95] Peter van Oosterom. The GAP-tree, an approach to "on-the-fly" map generalization of an area partitioning. *GIS and Generalization: Methodology and Practice*, pages 120–132, 1995. [p. 32]

[vR89] Jan W. van Roessel. An algorithm for locating candidate labeling boxes within a polygon. *The American Cartographer*, 16(3):201–209, 1989. [p. 154]

[VTW12] Mikael Vaaraniemi, Marc Treib, and Rüdiger Westermann. Temporally coherent real-time labeling of dynamic scenes. In *Proceedings of the 3rd International Conference on Computing for Geospatial Research & Applications (COM.Geo'12)*, pages 17:1–17:10. ACM, 2012. [pp. 17, 20, 73, 87, 110, 119, 122, 132, 135, 136, 137, and 144]

[WB05] Daniel Wigdor and Ravin Balakrishnan. Empirical investigation into the effect of orientation on text readability in tabletop displays. In Hans Gellersen, Kjeld Schmidt, Michel Beaudouin-Lafon, and Wendy E. Mackay, editors, *Proceedings of the 9th European Conference on Computer Supported Cooperative Work (ECSCW'05)*, pages 205–224. Springer-Verlag, 2005. [pp. 15, 83, and 119]

[WD98] Robert Weibel and Geoffrey Dutton. Constraint-based automated map generalization. In *Proceedings of the 8th International Symposium on Spatial Data Handling (SDH'98)*, pages 214–224. Taylor & Francis, 1998. [p. 32]

[WKvK$^+$00] Alexander Wolff, Lars Knipping, Marc van Kreveld, Tycho Strijk, and Pankaj K. Agarwal. A simple and efficient algorithm for high-quality line labeling. In Peter M. Atkinson and David J. Martin, editors, *Innovations in GIS VII: GeoComputation*, chapter 11, pages 147–159. Taylor & Francis, 2000. [p. 86]

[WWKS01] Frank Wagner, Alexander Wolff, Vikas Kapoor, and Tycho Strijk. Three rules suffice for good label placement. *Algorithmica*, 30(2):334–349, 2001. [p. 86]

[ZH06] Qing-nian Zhang and Lars Harrie. Real-time map labelling for mobile applications. *Computers, Environment and Urban Systems*, 30(6):773–783, 2006. [p. 56]