# Point Labeling with Sliding Labels in Interactive Maps

Nadine Schwartges, Jan-Henrik Haunert, Alexander Wolff, and Dennis Zwiebler

**Abstract** We consider the problem of labeling point objects in interactive maps where the user can pan and zoom continuously. We allow labels to slide along the point they label. We assume that each point comes with a priority; the higher the priority the more important it is to label the point. Given a dynamic scenario with user interactions, our objective is to maintain an occlusion-free labeling such that, on average over time, the sum of the priorities of the labeled points is maximized. Even the static version of the problem is known to be NP-hard.
We present an efficient and effective heuristic that labels points with sliding labels 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 screen space. With this data structure we were able to double the frame rate when rendering maps with many labels.

**Key words:** Dynamic maps, Interactive maps, Automated map labeling, Sliding labels, Point labeling

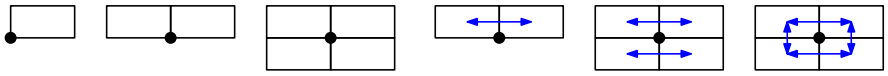
N. Schwartges (✉) · A. Wolff · D. Zwiebler
Chair of Computer Science I, University of Würzburg, Germany
URL: http://www1.informatik.uni-wuerzburg.de/en/staff

D. Zwiebler
e-mail: dennis.zwiebler@freenet.de

J.-H. Haunert
Institut für Geoinformatik und Fernerkundung, University of Osnabrück, Germany
URL: http://www.igf.uos.de/en/institute/staff



J. Huerta et al. (eds.): AGILE 2014, *Connecting a Digital Europe Through Location and Place*, LNG&C, pp. 295–310, 2014.



**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)

**Fig. 1** Examples of common labeling models: fixed-position models (a) and slider models (b)

## 1 Introduction

In navigation systems and online map services, map objects (such as cities) are typically annotated by labels (such as city names) in order to convey information about the map objects. While it is desirable to place many labels, it is difficult to do so since labels must not overlap each other. Many interactive 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 interacts with the map.

Map labeling is a classical problem in cartography. Cartographers such as Alinhac (1962) or Imhof (1975) have given rules for good label placement. Partially based on these rules, computer scientists have suggested many map-labeling algorithms in the 1980's and 1990's, especially for point objects. For practical purposes, the static point-labeling problem can be considered solved. Point labeling requires a *labeling model* that defines possible label positions. There are two types of such models. In *fixed-position models*, each label is restricted to a discrete set of positions relative to the point it labels; see Fig. 1(a). In *slider models* (Van Kreveld et al. 1999), each label can be placed at any position such that (a certain part) of its boundary touches the corresponding point; see Fig. 1(b). Each feasible placement of a label is called a label *candidate*. Usually, every 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 points.

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 $P'$ of $P$ and, for each point $p$ in $P'$, a label $\ell(p) \in L(p)$ such that no two labels overlap and the sum $\sum_{p \in P'} w(p)$ is maximized. The case of axis-aligned rectangular labels has been studied from a theoretical point of view. For fixed-position models, this problem is known as *maximum independent set* in weighted rectangle intersection graphs, which is known to be NP-hard (Fowler et al. 1981). There are, however, some approximation algorithms for the unweighted (Agarwal et al. 1998; Chalermsook and Chuzhoy 2009) and for the weighted case (Adamaszek and Wiese 2013; Erlebach et al. 2005). For slider models, too, the problem is known to be NP-hard (Poon et al. 2003), even for the most restricted slider model, the *one-slider model* (1S), where the bottom edge of the label must touch the corresponding point; see Fig. 1(b). For the weighted case and rectangular labels of equal height,

Erlebach et al. (2009) have given a *polynomial-time approximation scheme* (PTAS), that is, a $(1+\varepsilon)$-approximation algorithm (for any $\varepsilon > 0$).

**Our Model** In this paper, we are interested in a dynamic setting where the user can interact with the map by panning and zooming continuously. We consider a time interval $[0,T]$ in which the user interacts with the map. Current screens are redrawn repeatedly; the content of the screen between two updates is called a *frame*. Accordingly, we discretize the time interval into a sequence $t_1,\ldots,t_n$ (with $t_1 = 0$ and $t_n = T$) of points in time, which correspond to frames. At any time $t_i$, the user can see a rectangular region $R_i$ of the plane, the *view*. When the user pans, $R_i$ is translated; when the user zooms in or out, $R_i$ is scaled accordingly.

Now we can define the dynamic point-labeling problem DYNAPOINTLAB. For each $i = 1,\ldots,n$, let $P'_i$ be the subset of points in the view $R_i$ that are labeled 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 $W_i = \sum_{p\in P'_i} w(p)$. Then we define the overall quality of a dynamic label placement to be the quality, averaged over all frames: $\sum_{i=1}^{n} W_i/n$. Note that DYNAPOINTLAB generalizes STATPOINTLAB, which corresponds to the restriction to a single frame ($n = 1$) and a large enough view $R_1$.

There are, however, two further requirements for interactive maps, which were introduced by Been et al. (2006). They argued that in a *consistent* dynamic map labeling, labels should neither jump nor flicker (pop). In order to guarantee consistency, they disallowed labels to move at all and, when zooming, they insisted that a label is visible in at most one scale interval, the label's *active range*. Been et al. (2010) adopted the same rules and gave approximation algorithms for various special cases of the resulting (unweighted) optimization problem where the sum of the lengths of the active ranges is to be maximized. This is a continuous version of the objective function that we adopted above.

In this paper, we take a somewhat more pragmatic standpoint. We do allow labels to move. Still, our labels do not jump since we assume the one-slider model and our frame rates are high enough to ensure a smooth-looking movement when labels "slide". We do not, however, guarantee that labels don't flicker. We mitigate the problem for the map user by introducing a simple *waiting function* that suppresses labels for about 30 frames (that is, between 0.5 and 4 seconds) after they disappear.

As in most previous work, we assume that labels are axis-parallel rectangles. We decided to adopt the one-slider model due to a result of Van Kreveld et al. (1999) who compared various fixed-position and slider models using the same simple greedy algorithm for static point labeling. They found that a slider model yields about 15% more labels than the corresponding fixed-position model (on real-world data).

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 (RB/RE) trains of Deutsche Bahn[1] (German Railways) or browsers for large images of crowds that can be tagged[2]. If efficiency is less important, however, we

[1] http://bahn.de/zugradar, accessed Feb. 6, 2014

[2] http://www.u2.com/gigapixelfancam/, accessed Feb. 7, 2014

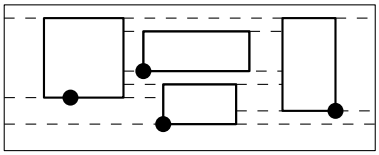


**Fig. 2** A rectangulation of a set of labels within a bounding rectangle $R$ (the view) is a subdivision of $R$ into labels and empty rectangles

suggest extending our algorithm to improve the quality of the labeling. For example, like Harrie et al. (2005) and Zhang and Harrie (2006), we could rule out positions where labels obscure other map objects or, following a rule of Imhof (1975), we could define preferences for certain label positions.

Similar to the algorithm of Zhang and Harrie (2006), our heuristic proceeds incrementally. It repeatedly goes through all points in the view and tries to label each unlabeled point, one at a time. Other than the algorithm of Zhang and Harrie, however, our algorithm may push away labels that have already been placed in order to make space for a new label. We use a geometric data structure that allows us to efficiently predict collisions when pushing labels. We may intuitively think of the labels as vessels drifting in water. At any time and for any label we need to know the neighbors of that label since these neighbors are the labels that are the possible counterparts for a collision. Exactly that problem, maintaining the adjacency relationships of moving vessels for collision avoidance, can be solved with a kinetic Voronoi diagram (Goralski et al. 2007). 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. Instead, we show how to use a *rectangulation* (see Fig. 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 (De Berg et al. 2008) where all trapezoids are rectangles. A rectangulation of the labels can be obtained by shooting horizontal rays from the top and bottom edges of the labels.

**More Related Work** Labeling interactive maps or 3d scenes is a relatively new research topic. When a user interacts with a 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 (2006) presented such a "memoryless" algorithm. Their labeling model doesn't insist that a label touches its point. To help the user understand the label–object association, they connect labels and objects with line segments, so-called *leaders*. Their approach runs in real-time. Mote (2007) introduced an algorithm for labeling points in interactive maps using the 4P 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. More recently, Luboschik et al. (2008) gave a heuristic for the problem of maximizing the number

of placed labels using the 4S labeling model as well as 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.

Gemsa et al. (2011b) considered the problem of maximizing the total length of the active ranges when labels are allowed to slide horizontally and the points are restricted to lie on the $x$-axis. The authors have presented an efficient PTAS for this problem. In order to support consistent labeling when users interactively rotate a map, Gemsa et al. (2011a) recently extended the idea of active ranges of scales to active ranges of rotation angles. Similarly, Gemsa et al. (2013) introduced active ranges of time, assuming that the user follows a pre-computed trajectory and that the viewport is centered on the user and oriented in the direction of movement.

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 two-dimensional active ranges, but, since current map viewers allow for zooming, rotating, panning, *and* tilting, we doubt that interactive labeling can be solved with the help of pre-computed active ranges alone. On the other hand, current algorithms that do not use preprocessing accept labels that flicker. Our approach with sliding labels, a waiting function, and a geometric data structure in the background can be seen as a compromise between these two worlds.

**Our Contribution** We use the dynamic rectangulation data structure mentioned above to design an interactive algorithm for DYNAPOINTLAB (see Sect. 2) and suggest ways to speed up this algorithm (see Sect. 3). Finally, we present some experiments with real-world data (see Sect. 4) and conclude the paper (see Sect. 5). A video that shows a result of our method can be found online [3].

## 2 Incremental Algorithm

In interactive maps, new labels can appear whenever the user interacts with the map. To avoid that labels flicker, we build and maintain our labeling and the corresponding rectangulation *incrementally* and use a waiting function (see Sect. 3.1). One incremental step roughly works as follows (also see Alg. 1). First, we locate the new point in the rectangulation. Next, we try to place its label such that it does not overlap other labels. This may imply that some labels may have to be pushed away or to be deleted. If the cost 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 following, we go through each of these steps in more detail.

[3] http://lamut.informatik.uni-wuerzburg.de/dynapointlab.html

**Alg. 1** IncrementalAlgorithm

**foreach** *point p to be labeled* **do**
- determine the rectangle 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
- slide $\ell(p)$ from its rightmost to its leftmost position
- combine the two sliding directions in order to determine a good position $\ell^*(p)$ for $\ell(p)$
- **if** *labeling p increases the total weight of the labeling* **then**
  - place $\ell(p)$ at $\ell^*(p)$
  - fix the rectangulation

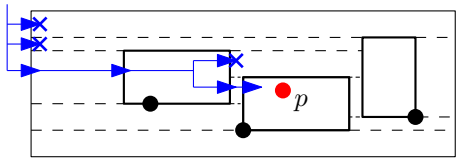



**Fig. 3** Illustration of the point-location algorithm. The point $p$ is the reference point of the label to place

## *2.1 Algorithm for 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 (De Berg et al. 2008). 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 Fig. 3. Let $p$ be the reference point to be labeled and let $y(p)$ be the $y$-coordinate of $p$. 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 yes, we are done. Otherwise, we go right. As each rectangle knows its (unique) right neighbor label $\ell$, we can easily test whether $\ell$ contains $p$. If not, we continue the search from $\ell$ 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 $O(\sqrt{n})$, where $n$ is the current number of labels in the view.

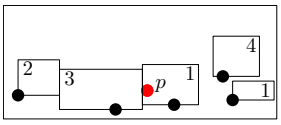
## 2.2 Algorithm for Sliding Labels

With the help of the point-location algorithm, we know the element of the rectangulation that contains the point $p$ to label. We next determine the final label position $\ell^*(p)$ of the label $\ell(p)$. In order to save running 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 over the priorities of all removed labels is as small as possible; the priority of a label $\ell(p)$ is the same as the priority 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 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 priority 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 Fig. 4. Only the final decision is visible to the user.

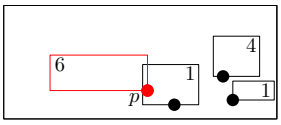
First, we set the label $\ell(p)$ to its leftmost position. We ignore all labels whose reference points lie to the left of $p$ (we will correct this error by combining the two sliding directions). Next, we determine *clusters* of labels. To this end, we use a directed *contact graph* whose vertices are the labels that are currently visible. There is an edge between the vertex $\ell(p)$ and each vertex whose label overlaps $\ell(p)$ as well as between two vertices if the boundaries of their labels touch sideways. We direct an edge $(\ell(u),\ell(v))$ such that $x(u) < x(w)$. To complete, a cluster $c(s)$ is the set of vertices that can be reached by a (source) vertex $\ell(s)$; see Fig. 5.

Assume that $\ell(p)$ is overlapped. 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 sliding clusters until $\ell(p)$ is occlusion-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 priority that lies between $\ell(p)$ (excluding) and a *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 priority $w(p)$ 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 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. We repeat building and sliding clusters and removing blocking labels until $\ell(p)$ is occlusion-free.

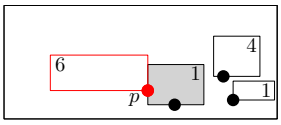
As soon as $\ell(p)$ is occlusion-free, we repeat the entire process with the objective that $\ell(p)$ reaches its right-most position, that is, we slide $\ell(p)$ within its cluster $c(p)$. To this end, we modify the process as follows: we use $c(p)$ instead of $c(s)$; we use a cost function and stop sliding to the right instead of rejecting $\ell(p)$ due to priorities.



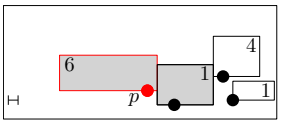

**a** the point $p$ to be labeled appears



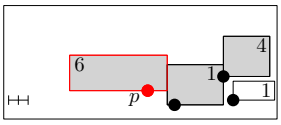

**b** set $\ell(p)$ to its leftmost position; neglect labels with reference points left to $p$



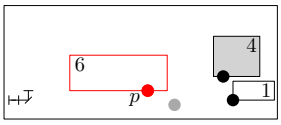

**c** slide overlapping label to the right to make $\ell(p)$ occlusion-free



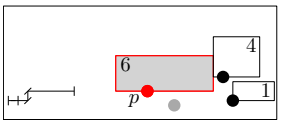

**d** $\ell(p)$ is occlusion free; slide cluster; record amplitude (left)



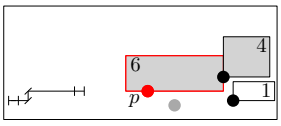

**e** slide cluster; record amplitude



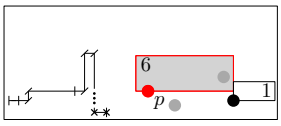

**f** raise blockade (uttermost position); record costs; some labels slide back



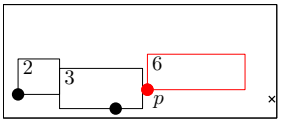

**g** slide further



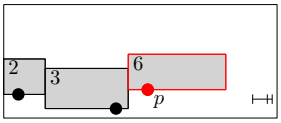

**h** slide further; next, raise blockade and slide further



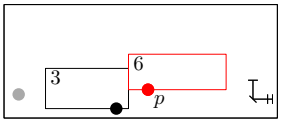

**i** set cost function negative due to priorities; we are done



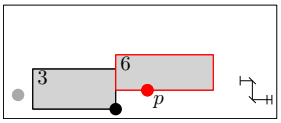

**j** from right to left: set $\ell(p)$ to its rightmost position



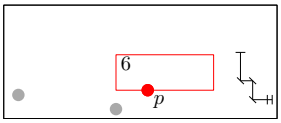

**k** slide



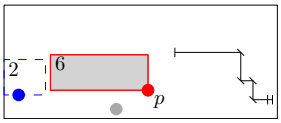

**l** raise blockade (view boundary)



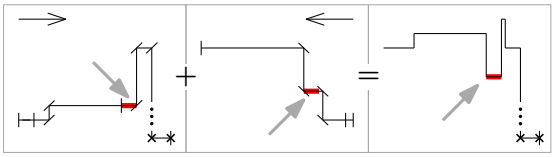

**m** slide further



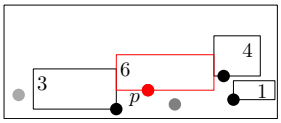

**n** raise blockade (uttermost position)




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



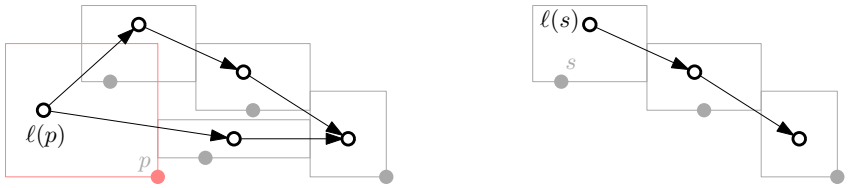
**p** cost functions, aggregated function, and minimum costs (arrowed)




**q** a feasible final configuration

**Fig. 4** Illustration of several steps of the algorithm for sliding labels. The point to be labeled is $p$. We annotated every label with its priority. The rectangulation is not shown

*[Hint: The content of this page differs from the original publication as we have corrected an error in the example above.]*




**a** a labeling and its corresponding contact graph **b** a cluster $c(s)$ with source $\ell(s)$

**Fig. 5** A contact graph and one possible cluster

Whenever we remove a label $\ell(q)$, we store the priority of $q$ and the current position of $p$ at $\ell(p)$, this is, the *amplitude*, in a cost function. 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 $-\infty$.

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. We sum up the cost functions. These aggregated costs represent the costs for sliding some labels to the right and some to the left. Next, we extract the minimum of the aggregated function. Remark that the minimum (normally) is bounded by two amplitudes. Indeed, each label position for $\ell^*(p)$ between these two amplitudes yields low costs. 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 $\ell^*(p)$.

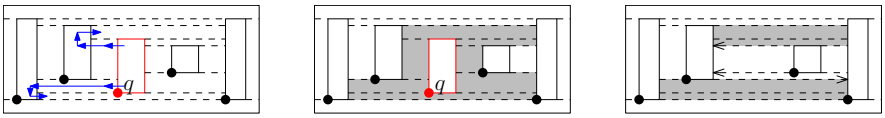
Note that our algorithm is a heuristic. In Fig. 4(o) we could re-insert the label on the left. So, we sometimes overestimate costs. This can finally result in the choice of a non-optimal amplitude, this is, we place fewer labels than possible. If we (try to) label unlabeled points in each frame, this error is quickly fixed.

## *2.3 Algorithm for 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 Fig. 6. 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




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

**b** rectangles to update are shaded

**c** lengthen lines; rectangles to merge shaded

**Fig. 6** Illustration of several steps of the algorithm for updating the rectangulation. The label to remove is $\ell(q)$

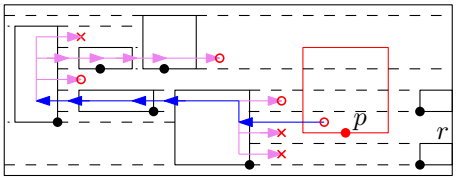
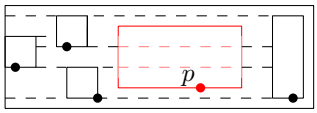


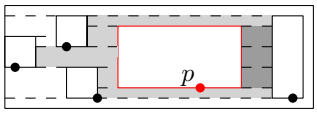

**Fig. 7** Illustration of the search originating from $r$ for detecting overlapped rectangles; a circle indicates an overlap with the label $\ell^*(p)$; a cross indicates the end of a search path

Fig. 6(a). Now, the set of neighbors of $\ell(q)$ is complete; see Fig. 6(b). We remove $\ell(q)$ and extend the horizontal edges of its neighbor rectangles to close the gap left by $\ell(q)$; see Fig. 6(c). As the number of empty rectangles influences the running 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 Fig. 7. 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 Fig. 8 for the result) we need to merge rectangles that are vertically adjacent to each other and have the same left and right neighbor.




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




**b** shaded rectangles were built; dark shaded rectangles must be merged

**Fig. 8** How to update the rectangulation if we place the label $\ell^*(p)$

## 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 function*; this is, we wait several frames until we try to label a certain reference point again. Furthermore, we discuss how to predict the point in time at which we have to trigger an update of the rectangulation.

### *3.1 Waiting Function*

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

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. For this, 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 $W$ and $|W|/f$ is an empirical value.

Recall that the algorithm for sliding labels does not re-insert labels; see Fig. 4(o). Applying the waiting function, 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. Indeed, the waiting function can cause quickly changing labels. Consider a labeling to which we can place $\ell(p)$ in frame $f$. In $f+1$ a more important label makes $\ell(p)$ disappear. We can continue this. Thus, each of these 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 priority of $\ell(p)$ and decrease it little by little. This approach has the advantage that we place labels with a much higher priority than $\ell(p)$ earlier than labels that are only slightly more important.

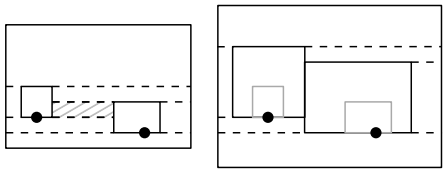


**Fig. 9** From left to right: If the user zooms out, the hatched rectangle collapses horizontally

### *3.2 Predicting Changes of the Rectangulation*

When a user pans or zooms the map, we need to update the rectangulation.

For panning operations, it is easy to predict the *event points* at which changes will be necessary. The 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 Sect. 2 in just the same way as for any other point. 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 crosses the boundary. Again, we can apply the algorithm of Sect. 2. After each update of the rectangulation, we compute new event points.

While the user zooms the map, we require that each label keeps its size on the screen. More precisely, labels have to shrink with respect to real-world coordinates while the user zooms in and labels have to grow while the user zooms out. Certainly, while zooming the map, empty rectangles can collapse and the $y$-order of the top edge of a label and the bottom edge of another label (nearby) can change, see Fig. 9. This makes the prediction of event points and a local update while zooming rather difficult. Therefore, in our current implementation, we recompute the rectangulation in each frame if the user zooms the map. An interesting question for future research is whether we can speed up our method by predicting changes of the rectangulation that are caused by a zooming operation.

## 4 Experiments

We have implemented the incremental algorithm from Sect. 2 using a rectangulation and a waiting function (Sect. 3.1). To estimate the value of our algorithm, we compared 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.

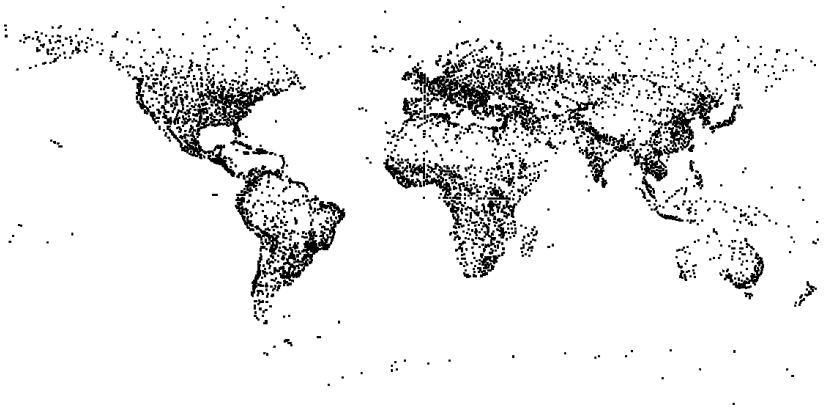


**Fig. 10** Point set of world map that we used in our experiments

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

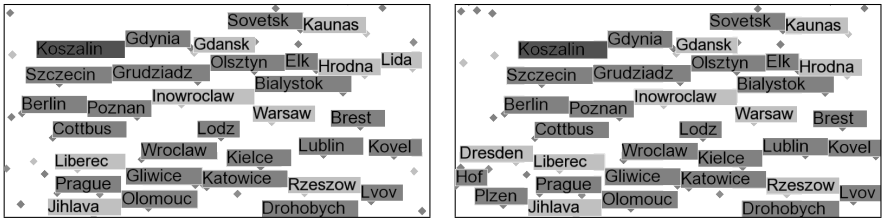
For our implementations, we used C++ with OpenSceneGraph 3.0[4]. We executed our experiments on a Windows 7 system with a 3.3-GHz AMD triple-core processor and 8 GB of RAM, applying the Microsoft Visual Studio 2010 Ultimate compiler in 32-bit release mode. The complete code has about 12,300 lines. For our tests, we used a world map from Natural Earth[5] providing 7,322 cities with priorities; see Fig. 10. We used priorities 1 (unimportant) to 4 (most important). 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. Additionally, we defined multi-interaction paths that pan and zoom in and zoom out. 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 made the resulting video available online, from the url referenced at the end of Sect. 1. Figure 11 shows two snapshots of a panning interaction.

To the resulting 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 function of $f = 30$ 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 the priority 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.

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

[5] http://www.naturalearthdata.com/, accessed Nov. 28, 2013




**a** initial labeling **b** labeling after panning to the left

**Fig. 11** A labeling before (a) and after (b) panning the map. New labels appeared at the left boundary of the view; at the right boundary, a label vanished. On the lower right, Lvov pushed Rzeszow, Rzeszow pushed Katowice, and so on

**Table 1** Quality: average value of the objective function (sum of the priorities over the labeled points on the screen)

| | | rectangulation | | | |
|---|---|---|---|---|---|
| | | $f=0$ | | $f=30$ | |
| | $\varnothing\lvert P'\rvert$ | 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 |

**Table 2** Speed: average frame rates (in FPS) of camera paths

| | | naive | | | | rectangulation | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $f=0$ | | $f=30$ | | $f=0$ | | $f=30$ | |
| | $\varnothing\lvert P'\rvert$ | 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 |

$f$ denotes the waiting interval (in frames); $\varnothing\lvert P'\rvert$ is the average number of labeled points on the screen; 1P and 1S are our labeling models; *rectangulation* and *naive* are our algorithms

For each frame, we recorded the sum of weights over all labeled points. We summed up the weights over 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 averaged quality; see Table 1. Additionally, we averaged the total number of frames over the processing time in order to compute the *frame rate* in frames per second (FPS); see Table 2.

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 also observed that our algorithms yield different results with regard to the averaged weight and FPS 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 1 shows only the quality results for the

original algorithm. Moreover, the results for zooming in only slightly differ from the results for zooming out (the inverted path). Thus, we averaged the results for zooming out.

We conclude that, using the slider model, our algorithm yielded an improvement of 30–50% in the labeling quality with respect to the algorithm using the fixed-position model; see Table 1. Second, we point out that the rectangulation-based approach increased the frame rate by up to 40% if the screen contained a large number of labels; see Table 2. If we additionally applied a waiting function of 30 frames, the frame rate for small point sets increased by about 15%. For large point sets, it sometimes doubled. The maximum loss in quality was 18%. When we applied a waiting function of 60 frames, to our surprise, the frame rates increased by at most 2 FPS whereas the quality dropped by up to 30%. Therefore, we do not show the details concerning the longer waiting function in Tables 1 and 2.

## 5 Conclusion and Future Work

In this work, we have described an algorithm that labels points in interactive maps using a slider model. To speed up our algorithm, we used a rectangulation data structure and a waiting function. 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, that is, in some cases, it doubled.

In the future, we plan to implement the prediction of changes in the rectangulation. Further, we want to analyse the cost of our current simplistic point-location strategy. Will it be worthwhile replacing it with a dedicated dynamic point-location data structure? It would also be interesting to deal with rotations and 3D environments where the view can be tilted or to study how users cope with the additional cognitive load of sliding labels. Ooms et al. (2009) showed that when panning horizontally, users did not react significantly to certain differences in the labeling.

## References

Adamaszek A, Wiese A (2013) Approximation schemes for maximum weight independent set of rectangles. In: Proc. 54th Annu. IEEE Symp. Foundat. Comput. Sci. (FOCS'13), pp 400–409

Agarwal PK, van Kreveld M, Suri S (1998) Label placement by maximum independent set in rectangles. Comput Geom Theory Appl 11:209–218

Alinhac G (1962) Cartographie Théorique et Technique, Institut Géographique National, Paris, chapter IV

Been K, Daiches E, Yap C (2006) Dynamic map labeling. IEEE Trans Visual Comput Graphics 12(5):773–780

Been K, Nöllenburg M, Poon SH, Wolff A (2010) Optimizing active ranges for consistent dynamic map labeling. Comput Geom Theory Appl 43(3):312–328

de Berg M, Cheong O, van Kreveld M, Overmars M (2008) Computational Geometry: Algorithms and Applications, 3rd edn. Springer-Verlag, chapter 6

Chalermsook P, Chuzhoy J (2009) Maximum independent set of rectangles. In: Proc. 20th Annu. ACM-SIAM Symp. Discrete Algorithms (SODA'09), pp 892–901

Erlebach T, Jansen K, Seidel E (2005) Polynomial-time approximation schemes for geometric intersection graphs. SIAM J Comput 34(6):1302–1323

Erlebach T, Hagerup T, Jansen K, Minzlaff M, Wolff A (2009) Trimming of graphs, with application to point labeling. Theory Comput Systems 47(3):613–636

Fowler RJ, Paterson MS, Tanimoto SL (1981) Optimal packing and covering in the plane are NP-complete. Inform Process Lett 12(3):133–137

Gemsa A, Nöllenburg M, Rutter I (2011a) Consistent labeling of rotating maps. In: Dehne F, Iacono J, Sack JR (eds) Proc. 12th Int. Symp. Algorithms Data Struct. (WADS'11), Springer-Verlag, Lecture Notes Comput. Sci., vol 6844, pp 451–462

Gemsa A, Nöllenburg M, Rutter I (2011b) Sliding labels for dynamic point labeling. In: Proc. 23th Canadian Conf. Comput. Geom. (CCCG'11), pp 205–210

Gemsa A, Niedermann B, Nöllenburg M (2013) Trajectory-based dynamic map labeling. In: Cai L, Cheng SW, Lam TW (eds) Proc. 24th Annu. Int. Symp. Algorithms Comput. (ISAAC'13), Springer-Verlag, Lecture Notes Comput. Sci., vol 8283, pp 413–423

Goralski R, Gold CM, Dakowicz M (2007) Application of the kinetic Voronoi diagram to the real-time navigation of marine vessels. In: Proc. 6th Int. Conf. Comput. Inform. Syst. Indust. Manag. Appl. (CISIM'07), pp 129–134

Harrie L, Stigmar H, Koivula T, Lehto L (2005) An algorithm for icon labelling on a real-time map. In: Fisher PF (ed) Proc. 11th Int. Symp. Spatial Data Handling (SDH'05), pp 493–507

Imhof E (1975) Positioning names on maps. Amer Cartogr 2(2):128–144

van Kreveld M, Strijk T, Wolff A (1999) Point labeling with sliding labels. Comput Geom Theory Appl 13:21–47

Luboschik M, Schumann H, Cords H (2008) Particle-based labeling: Fast point-feature labeling without obscuring other visual features. IEEE Trans Visual Comput Graphics 14(6):1237–1244

Maass S, Döllner J (2006) Efficient view management for dynamic annotation placement in virtual landscapes. In: Butz A, Fischer B, Krüger A, Oliver P (eds) Proc. 6th Int. Symp. Smart Graphics (SG'06), Springer-Verlag, Lecture Notes Comput. Sci., vol 4073, pp 1–12

Mote KD (2007) Fast point-feature label placement for dynamic visualizations. Information Visualization 6(4):249–260

Ooms K, Kellens W, Fack V (2009) Dynamic map labelling for users. In: Cartwright W, Lopez P (eds) Proc. 24th Int. Cartographic Conf. (ICC'09)

Poon SH, Shin CS, Strijk T, Uno T, Wolff A (2003) Labeling points with weights. Algorithmica 38(2):341–362

Zhang Q, Harrie L (2006) Real-time map labelling for mobile applications. Computers, Environment and Urban Systems 30(6):773–783