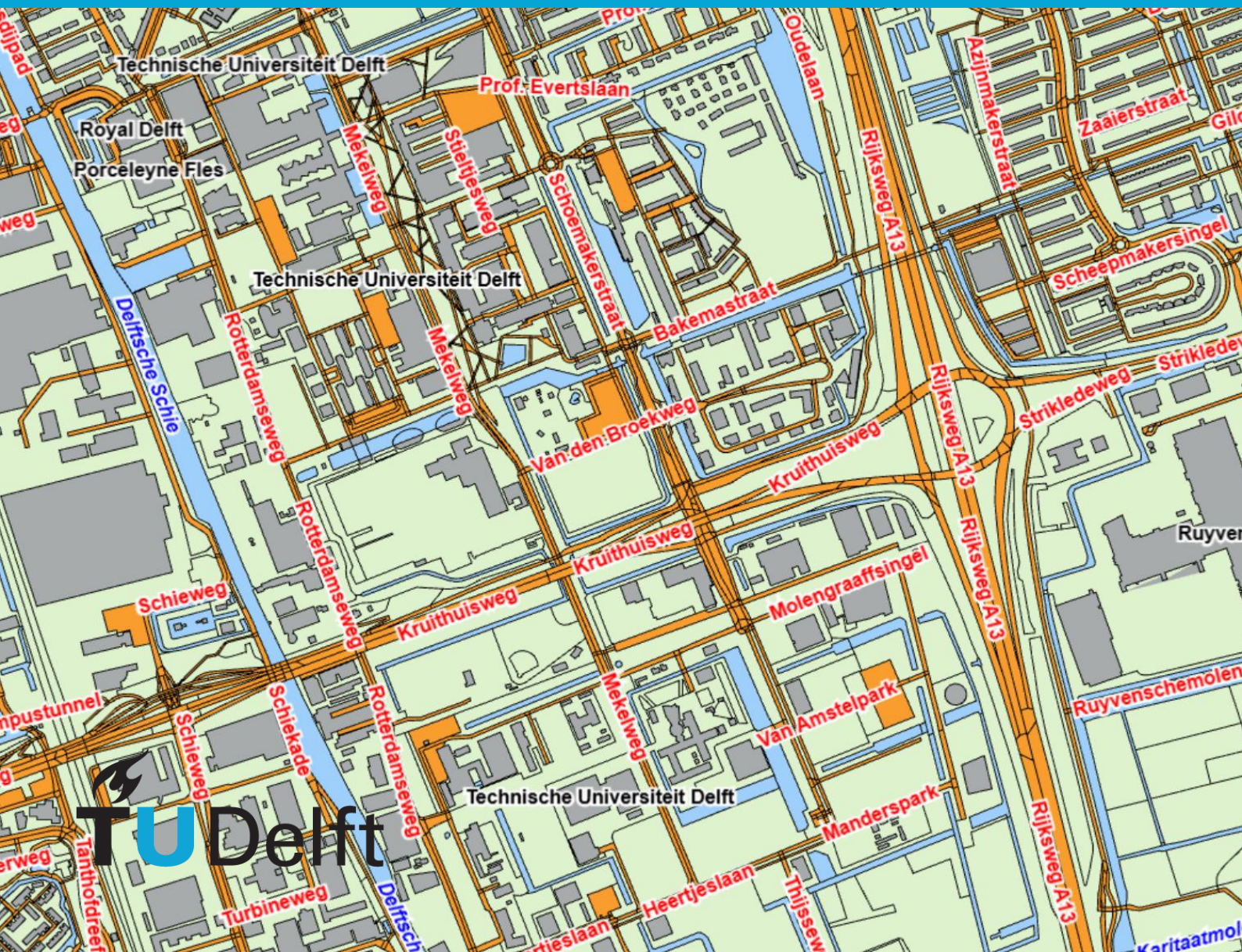
MSc thesis in Geomatics

Labeling Vario-scale Maps

Yan Gao

2025



MSc thesis in Geomatics

Labeling Vario-scale Maps

Yan Gao

June 2025

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of Master of Science in Geomatics

The work in this thesis was carried out in the:

GIS technology group
Delft University of Technology

Supervisors:	Dr.ir. Martijn Meijers
	Prof.dr.ir. Peter van Oosterom
Co-reader:	Dr. Ken Arroyo Ohori

Abstract

Map labels play an important role in helping users interpret geographic information. Traditional label placement algorithms are typically designed for static or fixed-scale maps, where labels appear at discrete, predefined zoom levels. However, vario-scale maps change content continuously during zooming, requiring labels to also transition smoothly to maintain readability and visual coherence.

This research explores how to place and adjust labels on vario-scale maps, ensuring legibility and usability throughout scale transitions. Due to limited existing work in this area, a set of requirements, both hard and soft, are defined to guide the methodology and evaluate results.

Different strategies are applied based on geometry types. For elongated features such as roads and rivers, label anchor points and orientations are derived from the medial axis (skeleton). For compact geometries such as buildings, the centroid and pole of inaccessibility are used to determine anchor points, while the minimum rotated bounding rectangle guides label rotation.

Two polygonization methods are developed using the tGAP structure: slice-based method extracts geometry at fixed scales and computes label positions per feature type; event-based method reacts to changes in geometry, computing label anchors whenever a change occurs. Label anchor points could be represented as continuous trajectories, supporting smooth interpolation. During interactive visualization, label positions, orientations, and opacities are interpolated across scales. To prevent clutter, real-time collision detection is implemented.

The methodology was applied to the TOP10NL dataset in the Delft region. Results showed that the event-based method allows more precise responsiveness to geometry changes but can produce occasional abrupt label movements. In contrast, the slice-based method offers more stable transitions but may cause unnecessary label shifts even when geometries do not change. Event-based labeling also resulted in a higher number of anchor points. While hard requirements were generally satisfied, some soft requirements were not fully met due to intentional trade-offs or implementation limitations.

Overall, this research introduces a novel framework for integrating dynamic labeling into vario-scale maps using the tGAP structure and demonstrates its feasibility through an interactive, real-time prototype with smooth label transitions and collision handling.

Online prototype available at: https://imyangao.github.io/Vario-scaleMapLabelingDemo/.

Source code on GitHub: https://github.com/imyangao/labelingVarioScaleMaps/tree/main and https://github.com/imyangao/addLabels2Map/tree/main.

Acknowledgements

Phew, here we are at the end of this thesis research.

The whole process has not been easy, there were times I felt stressed and anxious, and times I felt truly happy about making progress. But maybe that's what makes research so interesting: you never know what will happen as you explore. You learn and try things, you might succeed, you might fail, but most importantly, you realize you're not alone on this journey.

I truly want to express my gratitude to my supervisors. Throughout the process, they have been very supportive and helpful. Whenever I needed help, they were always there. Martijn is very strong in technical implementation aspects, whenever I ran into coding issues, his tips and help always guided me through. Peter has inspired me a lot with his passion for the Geomatics field. He can always bring up creative ideas and provide insightful guidance. I also want to thank Ken for his valuable input for my thesis.

A big thank-you to the friends around me, life would have been much harder without you. We're all struggling with our own life challenges, yet we still find ways to support each other. I cherish every relaxing chat, sunny walk, bubble tea afternoon, and delicious meal together.

Huge thanks to my boyfriend. He believes in me even more than I believe in myself, and his calm, easygoing life attitude helped to balance out my stress during this journey.

Lastly, I feel truly lucky to have all these people who stand by me, and I'm proud of myself for keeping on, even when things got tough.

Contents

Acronyms

1. Introduction

This opening chapter establishes the rationale and boundaries of the research. Section 1.1 explains the challenge and importance of labeling vario-scale maps. Section 1.2 then states the overall research objective, breaks it into specific sub-questions, and defines the technical scope and simplifying assumptions adopted throughout the thesis. Finally, Section 1.3 guides the reader through the remaining chapters.

1.1. Motivation

Map labels are important identifiers for geographic features, and their effective placement is important for the readability and usability of maps. Traditionally, label placement algorithms are designed for static or fixed-scale maps, where the map is viewed at discrete predetermined scales. However, vario-scale maps, which are maps that support a continuously changing scale, introduce unique challenges for labeling.

As the user zooms in or out smoothly, the map content changes in a continuous manner rather than in jumps, and the labels must adapt dynamically at every intermediate scale. For example, labels may need to reposition or even resize continuously to avoid overlapping with other labels and to prevent any label from abruptly disappearing or popping into view in a jarring way. If labeling is not handled properly in this context, the map can quickly become cluttered and confusing: overlapping labels or sudden label movements can distract and frustrate the user.

In a vario-scale scenario, every slight change in scale potentially affects which features are shown, how their geometries are represented, and consequently, which labels are displayed and where. Labels that were suitably placed at one scale might overlap or collide as the scale changes, or new space might become available for additional labels. Moreover, continuous scale change means there is no obvious "switch" point as in multi-scale maps; instead, labels should appear or disappear gradually to avoid distracting the user. Conventional cartographic labeling techniques, although effective for static maps and fixed-scale maps, are insufficient for vario-scale maps. They do not handle the continuous transitions of scale and can lead to issues like labels overlapping or flickering.

These challenges underscore the importance of developing new labeling approaches specifically suited for vario-scale maps. There is a need for strategies that enable labels to adjust their placement continuously and smoothly, ensuring that as the map scale changes, label positions and visibility update in a gradual manner.

Addressing this need is important to maintain map readability (so that text remains clear and unobstructed) and usability (so that users can navigate the map without distraction or confusion) throughout zooming and panning operations. And by introducing potential methods for label placement for vario-scale situations, this study may contribute to usage of vario-scale maps in practice.

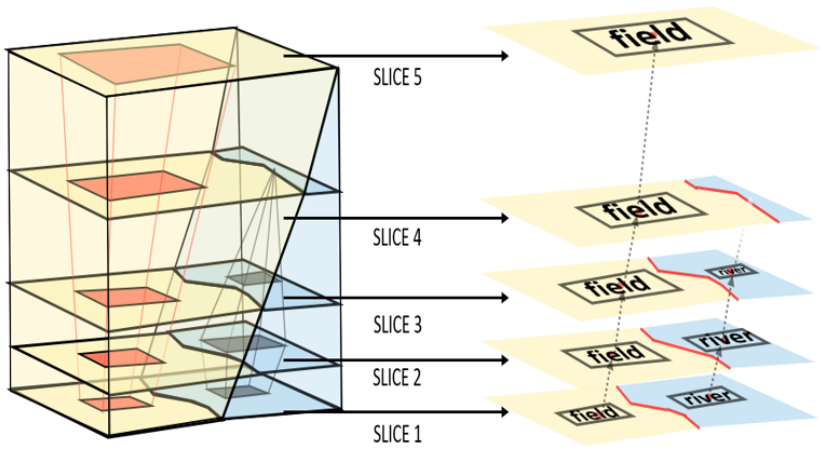


Figure 1.1.: Example of vario-scale maps with labeling

1.2. Research Objective and Scope

1.2.1. Research Objective

The aim of this research is to develop effective label placement strategies for vario-scale maps that allow labels to dynamically adjust with smooth transitions across scales. Ideally, labels on vario-scale maps should move without sudden jumps or abrupt disappearances (Figure 1.1).

The main research question of this thesis is:

How can labels be dynamically placed and adjusted on vario-scale maps to maintain readability, usability, and visual coherence across continuously changing scales?

Following sub-questions break down this broad question into specific components that address various technical and design challenges:

1. **Label placement requirements**: what are the hard and soft requirements for optimal vario-scale label placement?
2. **Optimal placement techniques**: how can the optimal positions for labels be determined for both elongated and more compact features, ensuring spatial alignment and visual clarity?
3. **Dynamic adjustments**: how can labels be smoothly transitioned across scales, minimizing positional shifts during scale changes, while maintaining readability and visual continuity?

4. **Data structure and retrieval**: how can label-related data be structured and stored efficiently to support dynamic rendering in vario-scale maps?
5. **Dynamic display**: how can label text be effectively placed on maps in a way that integrates with the underlying geographic features and supports continuous scale changes?

1.2.2. Research Scope

A simplifying assumption is adopted to manage the complexity of the problem. A label's visibility is tied directly to the visibility of its corresponding map feature in the vario-scale data structure. In practice, this means that when a geographic feature (for example, a polygon area) disappears from the map at a certain small scale (due to the continuous generalization process), its label is also removed as the feature disappears. The label set at any scale is therefore a set of labels for features that are present at that level of detail. The implicit importance is inherited from the vario-scale representation: larger or higher-priority features persist longer through scale changes, and so do their labels.

Further more, it should be notes that this research also excludes certain areas to maintain a focused approach. It excludes map rotation as a factor. All methods and evaluations will assume a fixed map orientation to simplify the scope and focus on other core challenges. It will not address labeling for multi-language or non-Latin scripts, limiting its scope to English characters. Additionally, the study is confined to 2D vario-scale maps and does not extend to 3D mapping.

Pre-existing label text data will be used, which refers to datasets where the labels (e.g., place names, street names, and points of interest) are already associated with their corresponding geometries in the map. This research does not focus on creating new label content (e.g., automatically generating street names or missing labels) but rather on optimizing the placement and rendering of existing labels across scales.

Furthermore, the research does not aim to develop new methods for text rendering but rather integrate existing techniques to achieve dynamic and visually coherent label placement. Curved labels that follow the shape of features (e.g., along rivers or roads) are also excluded; every label is treated as straight text aligned to a certain orientation.

1.3. Thesis Outline

The remainder of this thesis is organised into these chapters:

Chapter 2 reviews research related to vario-scale maps and existing different labeling methods. It begins with the tGAP and Space-Scale Cube (SSC) that enable vario-scale representations, then introduces both static and dynamic label-placement algorithms, identifying the open problems that motivate this study.

Chapter 3 introduces the hard and soft requirements that every design decision may follow, details label anchor points extraction for elongated and compact geometries, and illustrates two polygonization workflows (slice-based and event-based). It then explains how label trajectories are interpolated and how real-time collision handling is performed during interactive visualization.

Chapter 4 applies the methodology to TOP10NL dataset Delft region. It compares slice- and event-based anchors across scales, demonstrates smooth label transitions in an interactive viewer, and reflects on current limitations of the prototype.

Chapter 5 synthesizes the findings, highlights the thesis' main contributions to vario-scale maps labeling, and outlines certain aspects for future work.

2. Related work

This chapter reviews existing researches on vario-scale mapping and map labeling methods. The discussion begins with vario-scale techniques that enable smooth scale transitions, which may provide insights for integrating labeling methods that adapt to continuous zooming and panning. Next, labeling techniques for both static and dynamic maps are explored. This overview helps identify the main developments and gaps in the literature, providing a foundation for subsequent research.

2.1. Overview of Vario-Scale Maps

Vario-scale maps are designed to provide seamless transitions between map scales, enabling continuous zooming and panning without the abrupt changes characteristic of traditional multi-scale maps. This approach contrasts with traditional multi-scale maps, which rely on predefined levels of detail and can result in abrupt changes in map content when zooming in or out. Vario-scale maps achieve a smooth transition through data structures and algorithms that integrate geometry, topology, and scale information. This ensures smooth visual changes as one zooms in or out, avoiding the sudden appearance or disappearance of features that occurs when switching between fixed scale levels.

2.1.1. tGAP

One of the key frameworks enabling vario-scale maps is the topological Generalized Area Partition (tGAP). The tGAP is a vario-scale data structure designed to support map generalization across multiple levels of detail. It typically begins with a highly detailed planar partition of the map (at the largest scale available) and then progressively simplifies this partition by merging or eliminating less significant features (van Oosterom and Meijers [2011]). Each generalization operation (such as merging adjacent polygons or removing minor details) is recorded in a Directed Acyclic Graph (DAG), ensuring that all levels of detail are linked and accessible. The tGAP structure integrates both geometric and topological information, enabling efficient retrieval of representations at different scales. Figure 2.1 shows an example of a tGAP structure.

However, the classic tGAP implementation introduces discrete transitions between scales, which may lead to abrupt changes in the map, such as objects suddenly disappearing or boundaries shifting. These limitations are addressed in more advanced models, such as smooth tGAP and the space-scale cube (SSC), discussed in the following subsection.

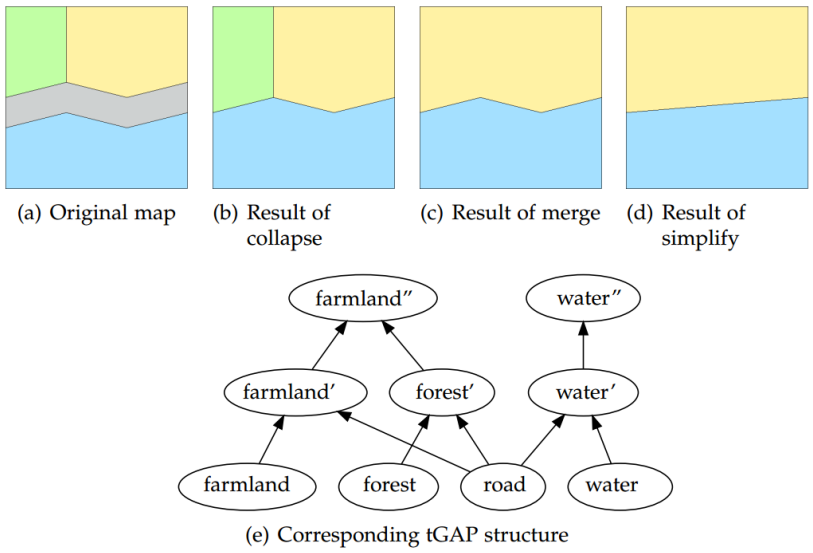


Figure 2.1.: Map fragments and corresponding tGAP structure (van Oosterom and Meijers [2012b])

2.1.2. SSC

Space-Scale Cube (SSC) is a 3D representation of geographic information that integrates the vertical dimension of scale into the data structure (van Oosterom and Meijers [2012b]).

In the Space-Scale Cube (SSC), 2D features from a map are extruded into a third dimension, where the height represents scale. Polygons are represented as 3D prisms, lines as vertical faces, and points as vertical lines (van Oosterom and Meijers [2012a]).

While the tGAP structure achieves a degree of vario-scale, its classic implementation introduces many small discrete transitions between scales, leading to abrupt changes in the map, such as objects suddenly disappearing or boundaries shifting, as shown in Figure 2.2.

To address these limitations, the smooth tGAP was introduced as an extension of the classic model. Smooth tGAP incorporates an explicit continuous scale dimension into the data structure, allowing gradual transformations of features rather than abrupt, stepwise changes. For example, objects shrink or grow progressively, and polygon boundaries adjust smoothly. These transitions are captured using tilted faces in a 3D space, creating a seamless representation of geographic features across scales. One example of a smooth tGAP structure is shown in Figure 2.3.

Because the geometry of smooth tGAP changes continuously along the vertical axis, a small change in the slice height (scale) causes only a small, smooth change in the resulting map features. Unlike the classic tGAP, which creates discrete snapshots at different scales, the SSC of smooth tGAP supports continuous representations. The comparison is shown in Figure 2.4.

Each 2D map can be derived by slicing the SSC horizontally at a specific scale, yielding a planar partition, while non-horizontal or tilted slices can produce mixed-scale maps.

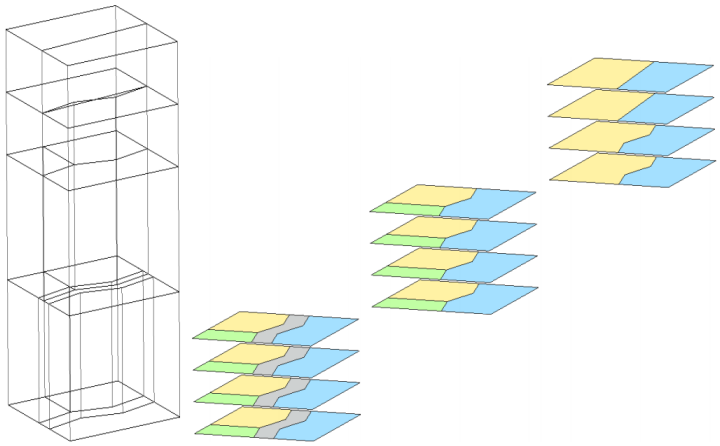


Figure 2.2.: SSC and map slices of the classic tGAP structure (van Oosterom and Meijers [2012b])

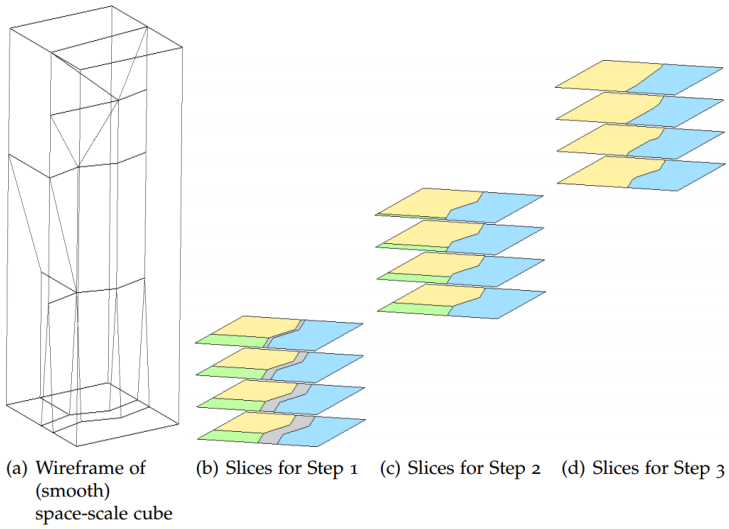


Figure 2.3.: SSC and map slices of the smooth tGAP structure (van Oosterom and Meijers [2012b])

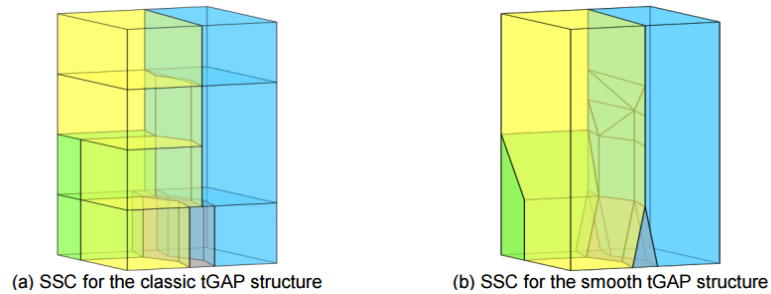


Figure 2.4.: Space-scale cube(SSC) representation in 3D (van Oosterom and Meijers [2012b])

In this research, only horizontal slices are considered to derive 2D maps at specific scales. Non-horizontal or tilted slices, which produce mixed-scale maps, are excluded to simplify the implementation of label placement algorithms. Mixed-scale maps introduce varying levels of detail within a single view, which can complicate label positioning as features from different scales may require different placement strategies within the same slice.

2.2. Label Placement Techniques

Effective label placement is important for map readability. This section reviews two categories of labeling techniques: static label placement (for fixed one scale maps) and dynamic label placement (for interactive maps), which may help to achieve smooth label transitions for vario-scale maps.

2.2.1. Static Label Placement

Static label placement techniques focus on optimizing label positions on fixed-scale maps, adhering to cartographic rules for clarity and spatial efficiency. Given a set of features (points, lines, areas) and their associated text labels, the goal is to place each label such that it is clearly readable and unambiguously associated with its feature, without overlapping other labels or symbols. While effective for static maps, these methods lack adaptability to dynamic scales, though their structured frameworks and optimization techniques may provide valuable insights for vario-scale labeling.

Edmondson et al. (Edmondson et al. [1996])proposed a general cartographic labeling algorithm that optimizes label placement for point, line, and area features on static maps. Their approach is structured into three key steps: candidate-position generation, position evaluation, and position selection. By using simulated annealing to minimize overlaps and maximize clarity, the algorithm produces visually appealing labels for dense, page-sized maps.

Van Roessel (van Roessel [1989]) introduced an algorithm for locating candidate labeling boxes within polygons, addressing challenges where centroid-based label placement results in awkward or overlapping positions. The method divides a polygon into horizontal strips, generates vertical segments within these strips, and identifies maximal candidate boxes that

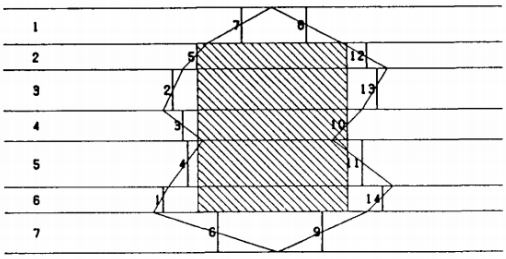


Figure 2.5.: Simple polygon with strips, vertical line segments and sample box (van Roessel [1989])

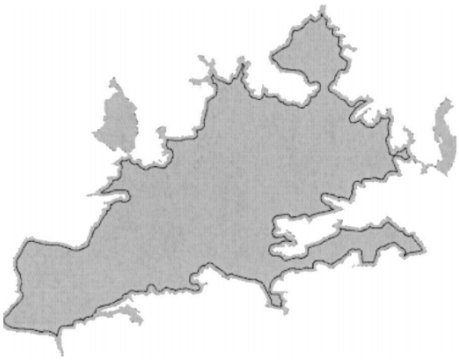


Figure 2.6.: Morphological erosion to extract the most important part (Barrault [2001])

are wholly contained within the polygon (Figure 2.5), allowing flexible label placement and repeating labels for large polygons.

Barrault (Barrault [2001]) also introduced a methodology specifically for labeling area features, addressing challenges such as complex shapes and overlapping constraints. The process incorporates morphological operations (erosions) (Figure 2.6) to simplify area boundaries, skeletonization to extract important geometric features (Figure 2.7), and a quality function to evaluate label placements based on their perceived coverage and conformity to the area shape.

Similarly, Dörschlag et al. (Dörschlag et al. [2003]) proposed an algorithm for placing objects, such as labels or diagrams, within map areas while maintaining cartographic quality. The approach involves a two-step process: erosion of the area boundary to identify feasible placement regions for objects and reduction of these regions to optimal positions using

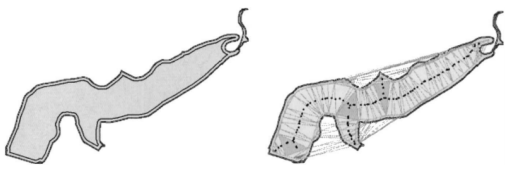


Figure 2.7.: Eroded area and its skeleton (Barrault [2001])

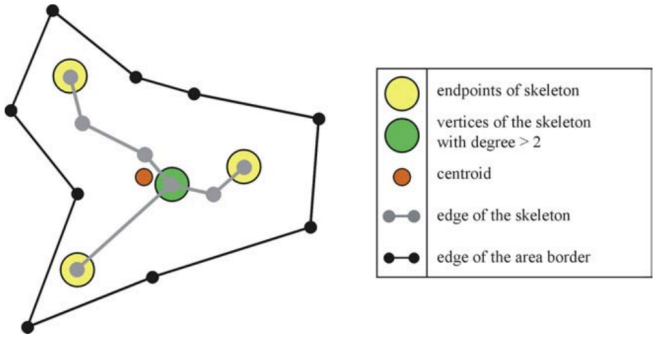


Figure 2.8.: Special points of the area and the skeleton (Dörschlag et al. [2003])

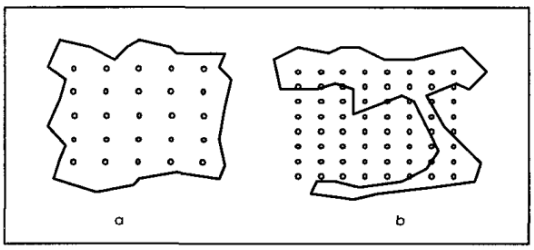


Figure 2.9.: Anchor points as candidates for the start and end points of the text's baseline (Pinto and Freeman [2006])

the skeleton of the eroded area (Figure 2.8). The scoring function evaluates candidate positions based on criteria like proximity to the area centroid and skeleton nodes with high connectivity.

Pinto and Freeman (Pinto and Freeman [2006]) introduced a feedback-based method for the automated placement of text labels within bounded regions on maps, emphasizing conformity to region size and shape. The method includes a two-step feedback process: an initial placement is generated based on anchor points within the region (Figure 2.9), and the placement is iteratively refined by evaluating it against quality criteria such as length, symmetry, clearance, and conformity. This iterative refinement allows the system to handle the complex and varied geometries more effectively than static algorithms.

Wolff et al. (Wolff et al. [2001]) proposed a simple and efficient algorithm for labeling linear features, such as rivers or roads, on maps. Their method creates a candidate strip along the polyline, ensuring that labels follow cartographer Imhof's ealier work (Imhof [1975]) for line labeling by satisfying hard constraints (minimum distance, curvature limits, and non-intersection) and optimizing soft constraints (proximity to the polyline, minimal inflections, and horizontal alignment).

Barrault and Lecordix (Barrault and Lecordix [1995]) developed an automated system for linear feature name placement that adheres to stringent cartographic quality criteria. Their methodology involves dividing roads into sections for labeling, generating candidate positions along straight parts of the road, and evaluating these positions based on intrinsic quality measures such as curvature, local centering, and background overlap. Simulated

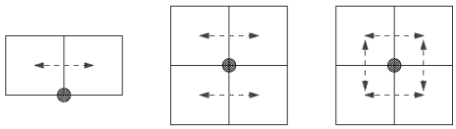


Figure 2.10.: Top-, two- and four-slider model (van Kreveld et al. [1999])

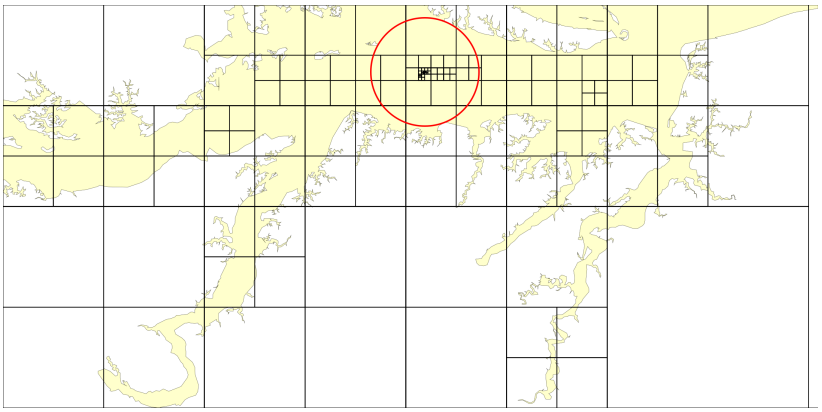


Figure 2.11.: Polylabel (Agafonkin [2024])

annealing is used to optimize label placement locally, ensuring regular spacing along roads while avoiding conflicts with other map features. Its focus on maintaining spatial harmony and addressing label repetition offers valuable insight.

Van Kreveld et al. (van Kreveld et al. [1999]) proposed algorithms for point labeling using sliding label models (Figure 2.10), which allow labels to continuously move along the edges of a rectangle instead of being restricted to a finite set of positions.

Strijk and van Kreveld (Strijk and Kreveld [2000]) extended the sliding label model for point labeling by introducing practical constraints such as obstacles and varying label heights. Their algorithm efficiently ensures that labels avoid overlapping with obstacles, such as boundaries, while maintaining proximity to the labeled point.

Inspired by the idea of pole of inaccessibility, which was first introduced by Vilhjalmur Stefansson (1920) to distinguish between the North Pole and the most difficult-to-reach place in the Arctic (Garcia-Castellanos and Lombardo [2007]), the pole of inaccessibility in a polygon is defined as the point inside a polygon that is furthest from its edges, maximizing internal clearance. A known practical implementation of this concept is the `polylabel` algorithm developed by Mapbox[1]. By recursively subdividing the search space and prioritizing candidate cells with greater distances from the polygon boundaries, the algorithm achieves a balance between computational speed and spatial accuracy, as shown in figure 2.11 (Agafonkin [2024]).

Static label placement techniques offer structured frameworks for maximizing the number of labels, optimizing label positions, minimizing overlaps, and ensuring cartographic clarity. Methods such as skeletonization, candidate position generation, and simulated annealing

[1] https://github.com/mapbox/polylabel

provide solid solutions for handling dense maps, complex geometries, and specific feature types (e.g., lines, areas, and points). However, these methods also have notable limitations. They are inherently designed for static, fixed-scale contexts and do not adapt dynamically to user interactions like zooming. In summary, while static label placement methods offer insights into optimal label placemens, their inability to handle continuous scale transitions and user interactions highlights the need for new approaches that support vario-scale maps.

2.2.2. Dynamic Label Placement

Dynamic label placement techniques focus on the challenges of real-time labeling in interactive maps, ensuring smooth transitions during zooming and panning. They highlight the importance of balancing computational efficiency with cartographic quality, providing insights for adapting dynamic principles to vario-scale map labeling systems.

Been et al. (Been et al. [2006]) introduced a framework for dynamic map labeling, addressing challenges of continuous zooming and panning while maintaining label consistency and interactive performance. They proposed desiderata for dynamic label consistency:

- Monotonicity of labels: labels should not appear, disappear, and reappear unpredictably as the user zooms in or out.
- Continuous transitions: label positions and sizes should change smoothly during panning and zooming.
- Pan consistency: labels should not unexpectedly vanish or appear during horizontal or vertical panning.
- History independence: the placement and selection of labels should depend solely on the current map state (scale and view), not the interaction path taken to reach it.

To address above challenges, the authors proposed an invariant point placement approach, where each label is associated with a fixed point on the map, ensuring that it stays in place relative to the feature being labeled. Labels maintain a constant screen size across scales, ensuring readability without abrupt shifts. This placement method leads to cone-shaped ”extrusions” in a space-scale representation, where the label’s visibility at different scales forms a continuous 3D cone, as shown in Figure 2.12. The idea of invariant point placements and active ranges offers a possible starting point for exploring how labels can appear and disappear in a user-friendly manner during continuous scale transitions.

The approach prioritizes computational efficiency and guarantees consistent label visibility during interactions. However, the method restricts label positions to avoid ”sliding” along features, which the authors consider an acceptable trade-off for performance and stability.

Building upon the earlier work, Been et al. (Been et al. [2010]) proposed the Active Range Optimziation (ARO) framework for consistent dynamic map labeling. The ARO problem focuses on selecting the optimal scale intervals during which labels remain visible (i.e., their ”active ranges”) while maximizing label presence across scales and avoiding conflicts. Their approach involves considering labels as geometric shapes (cones or rectangles) extruded across the scale dimension and proposes methods to approximate optimal active ranges using dynamic programming, greedy algorithms, and divide-and-conquer optimization techniques.

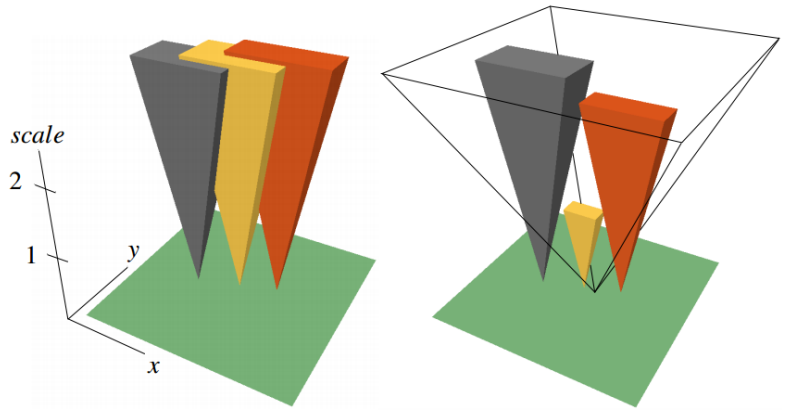


Figure 2.12.: Dynamic placements for three labels (without and with active ranges) (Been et al. [2006])

Schwartges (Schwartges [2015]) explored dynamic label placement for interactive maps, emphasizing real-time adaptability during user interactions like zooming and panning. In Section 3.3 of his thesis, the focus is on two greedy heuristics: shrinking-cones and growing-cones. The shrinking-cones method starts with full visibility for labels and iteratively reduces their active ranges to resolve conflicts, whereas the growing-cones method begins with minimal active ranges and gradually expands them until conflicts arise, and when it happens, the system removes labels in crowded areas to make the map easier to read (Figure 2.13). Both heuristics aim for fast computation and near-optimal solutions, balancing performance with cartographic quality.

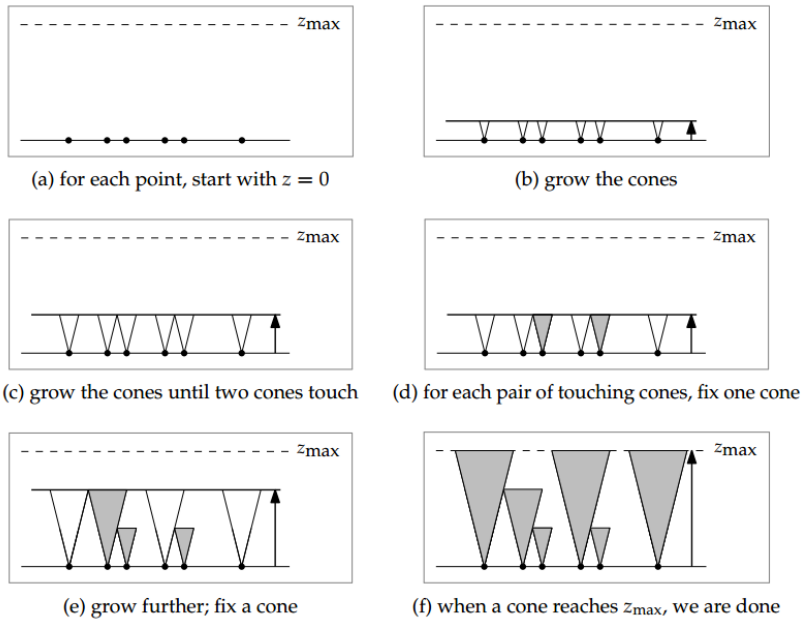
Both heuristics aim to maximize label presence over varying scales while maintaining computational efficiency, as exact solutions are computationally expensive for real-time applications. The experiments in this section also highlight that while greedy methods may not always yield optimal results, they produce near-optimal solutions with minimal computational overhead.

Dynamic label placement remains an active area of research, especially as interactive maps become ubiquitous, different techniques strive to maintain the readability and usability by managing label visibility ranges and positions continuously.

2.3. Open Questions from Literature

The literature review reveals several open questions about labeling methods for vario-scale maps. Addressing these questions is important for further advancing label placement techniques in continuous scale transitions:

- How can traditional static labeling methods be adapted to support the dynamic nature of vario-scale maps?
- How to adjust label placement smoothly for vario-scale maps?



(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 2.13.: Illustration of growing-cones heuristic (Schwartges [2015])

- What data structures and retrieval methods can best support the efficient rendering of dynamic labels, ensuring that labels appear at appropriate scales and positions?

These open questions underscore the necessity for novel methodologies tailored to labeling vario-scale maps, highlighting gaps that this research aims to address.

3. Methodology

In this chapter, the overall methodology for vario-scale map labeling is outlined (Figure 3.1). It starts with a statement of the hard and soft requirements that guide every design choice. It then illustrates how anchor points are derived for geometries, how faces on maps are polygonized across scales, and how those anchors are linked into continuous trajectories. The chapter also provides an overview of how label positions are interpolated, how label collisions are resolved and how labels are combined with underlying maps during visualization.

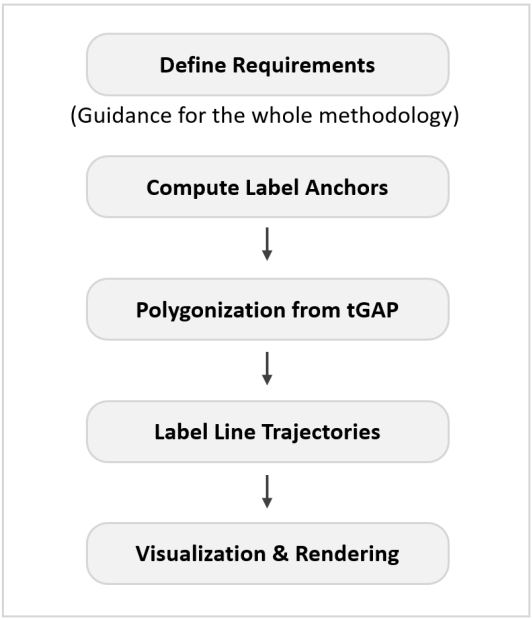


Figure 3.1.: Workflow of the proposed vario-scale map labeling methodology

3.1. Label Placement Requirements

Due to the lack of established guidelines for labeling vario-scale maps, we introduce a two-tier requirement framework (Figure 3.2) to support subsequent design decisions, grounded in both existing literature as introduced in Section 2.2.2 and insights from thesis supervisors. The **hard requirements** are non-negotiable: violating any one of them breaks the map's basic communicative purpose. The **soft requirements**, on the other hand, enhance cartographic quality but may be relaxed or omitted without rendering the map unreadable.

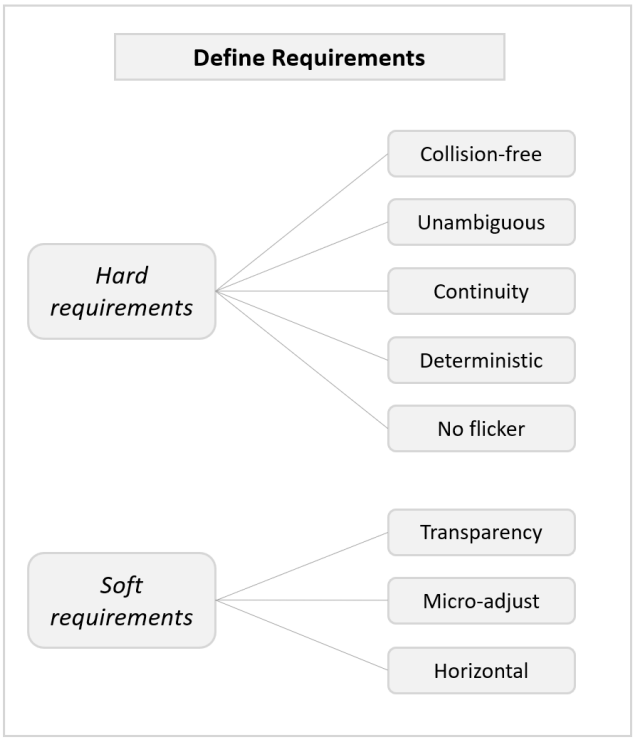


Figure 3.2.: Definition of hard and soft requirements

By separating these tiers, we obtain a clear direction for both implementation trade-offs and later evaluation.

3.1.1. Hard (Must-Have) Requirements

H1. Collision-free rendering: at any visible scale, no two label bounding boxes may overlap in screen space as overlaps obscure text and reduce readability.

H2. Unambiguous feature association: a label must be visually and semantically linked to exactly one map feature at any moment.

H3. Label continuity during continuous zoom: label anchors must move smoothly as the view scale changes; abrupt repositioning should be forbidden.

H4. Deterministic priority model: given the same view parameters, the label selection and placement must be consistent and repeatable.

H5. No flicker: labels should appear and disappear smoothly, without sudden flickers during interaction.

3.1.2. Soft (Nice-to-Have) Requirements

S1. Avoid half-transparent rest state: labels should not remain partially transparent at rest, as this can cause confusion or misinterpretation.

S2. Micro-adjust before drop: if two labels overlap by less than a certain percentage of their areas, attempt a slight nudge before removing the lower-priority label. This may increase label density without compromising clarity.

S3. Preserve horizontal orientation: when feasible, keep labels aligned horizontally to enhance legibility.

3.2. Label Placement

Placing descriptive labels inside polygonal features may require methods tailored to the characteristics of those features. In this section, we distinguish between two different types of geometries and use different label anchor points placement strategies for each: elongated geometries (e.g. roads or rivers, which are long and narrow) and compact geometries (e.g. building footprints) (Figure 3.3).

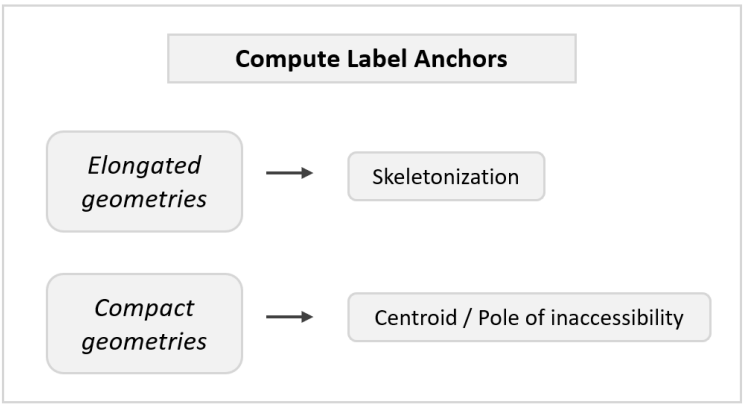


Figure 3.3.: Anchor computation methods for elongated and compact geometries

3.2.1. Label Anchors for Elongated Geometries

Elongated map features like roads, rivers, and canals span large extents, often stretching across large portions of the map. A single label placement for such features would be inadequate because it would fail to clearly identify the feature across its entire extent. Therefore, it's necessary to place multiple labels at various intervals for elongated geometries to consistently remind users of the feature's identity. To achieve distributed and visually coherent label placements, we apply a skeletonization technique inspired by the work of Dörschlag et al.Dörschlag et al. [2003], as discussed in Section 2.2.1.

Skeletonization refers to computing the medial-axis or "skeleton" of a polygon: a set of lines equidistant from the polygon's boundaries that capture its general shape and connectivity.

By placing labels along these centerlines, we ensure the text follows the feature's primary orientation and tend to stay well within its boundaries.

We use `Grassfire` library[1] to generate medial-axis-based skeleton, which results in linear representations reflecting the central axes of the elongated shapes.

The resulting skeleton lines form a graph structure where nodes represent intersections, and edges represent line segments (Figure 3.4). Nodes connecting multiple segments(points where three or more skeleton branches meet) - junction nodes - are identified to outline important intersections within the geometry. Primary paths between these nodes are then selected. As shown in Figure 3.5, small branches are ignored and main skeleton structure is kept.

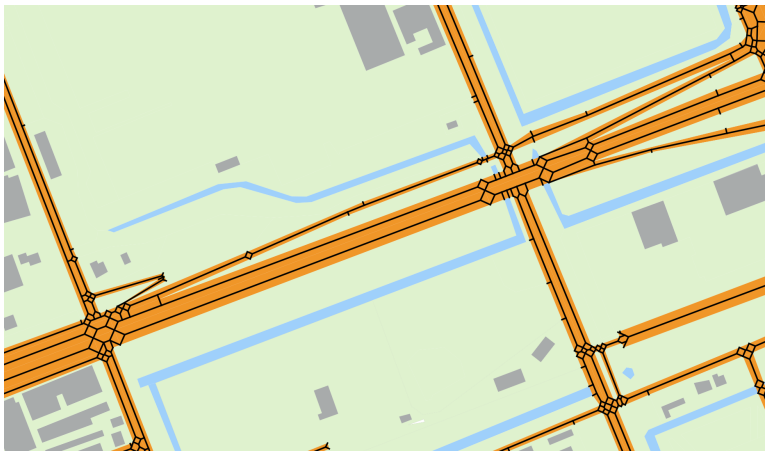


Figure 3.4.: Overview of skeletons(due to skeletonization, there are many small branches)

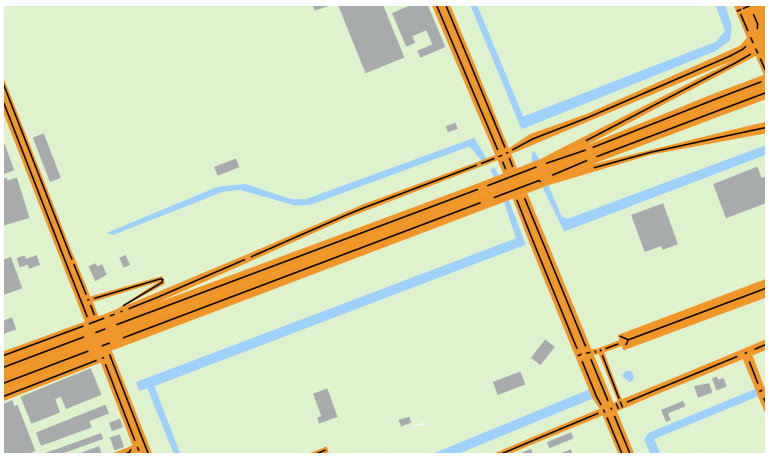


Figure 3.5.: Main skeleton extraction

Midpoints of the long segments (segments exceeding a certain threshold of the length of

[1]https://github.com/bmmeijers/grassfire

the longest segment) are selected as anchor points for placing labels along the geometry, ensuring the label appears multiple times at readable locations (as shown in Figure 3.6). Label orientation angles are calculated based on the rotation of the corresponding segments, ensuring alignment along the elongated geometry.

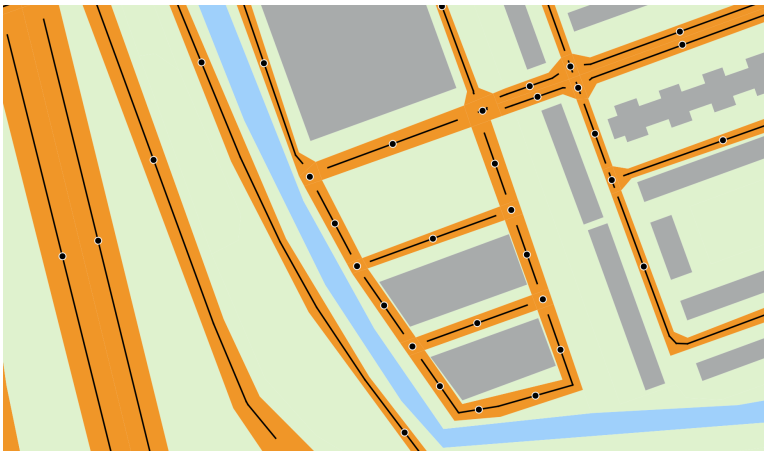


Figure 3.6.: Points represent label anchor points

Table 3.1 summarizes key parameters for the skeletonization process and their roles. Some parameters (e.g. length threshold fraction) could be tuned to generate label anchors for elongated shapes:

Parameter	**Description**
Junction degree threshold	Minimum degree for skeleton nodes to qualify as junctions. Degree $\geq$3 indicates meeting of 3+ skeleton branches (significant branching points).
Length threshold fraction	Fraction of longest skeleton path length used as cutoff. Segments shorter than certain percentage of max length are ignored to avoid labeling minor features.
Anchor point (placement)	Label anchor placed at midpoint of selected skeleton segment, centering label along interior line.
Label orientation angle	Text rotated to align with skeleton segment direction. Aligns label with feature's major axis.

Table 3.1.: Key parameters for skeleton-based label anchor points computation

3.2.2. Label Anchors for Compact Geometries

Compact geometries, such as buildings, generally occupy limited spatial extents. Unlike elongated features, placing multiple labels on these shapes would introduce unnecessary clutter and reduce readability. Therefore, we determine a single, clear anchor point per feature to ensure effective labeling without overwhelming the map (Figure 3.7).

In this research, we adopt a two-stage strategy to determine the label anchor point for compact geometries:

- **Primary approach**: we use the polygon's centroid as the label anchor. In most compact, convex shapes, the centroid offers a stable and central position for placing a label.
- **Fallback approach**: if the geometry is concave and the centroid lies outside the polygon, we instead use the pole of inaccessibility, which is the point within the polygon that is farthest from any edge.

We choose the centroid as the default because it provides a consistent anchor location and avoids instability. In contrast, the pole of inaccessibility, while optimal in terms of interior distance, can sometimes shift label anchor positions significantly with small geometric changes. Thus, we use it only as a fallback when the centroid is not suitable.

The pole of inaccessibility is computed using the `polylabel`, available via the `Shapely` library. The function `polylabel(polygon, tolerance)` approximates the pole of inaccessibility to a specified precision.

After determining the anchor point (either the centroid or the pole of inaccessibility), we compute the label orientation using the polygon's minimum rotated rectangle containing the polygon via Shapely's `polygon.minimum_rotated_rectangle`. This method generates the smallest area enclosing rectangle. From its edges, we identify the longer pair of sides and use their direction to define the label orientation. This aligns the label with the major axis of the shape.

Figure 3.7.: Label anchor points for buildings

3.3. Polygonization from tGAP

Polygonization from tGAP data can be achieved via two methods (Figure 3.8). In both cases, the goal is to derive the polygonal faces present at certain map scales from a tGAP data structure, which encodes edges and faces with the scale ranges (step_low, step_high) over which they exist, and to compute label anchor points for each face. Information about tGAP data schemas are described in Appendix B. In this section two methods are described in details: a slice-based approach and an event-based approach. Both methods rely on spatial database operations and ensure topological consistency of the polygonized faces. Each method produces a set of face anchor points across scales.

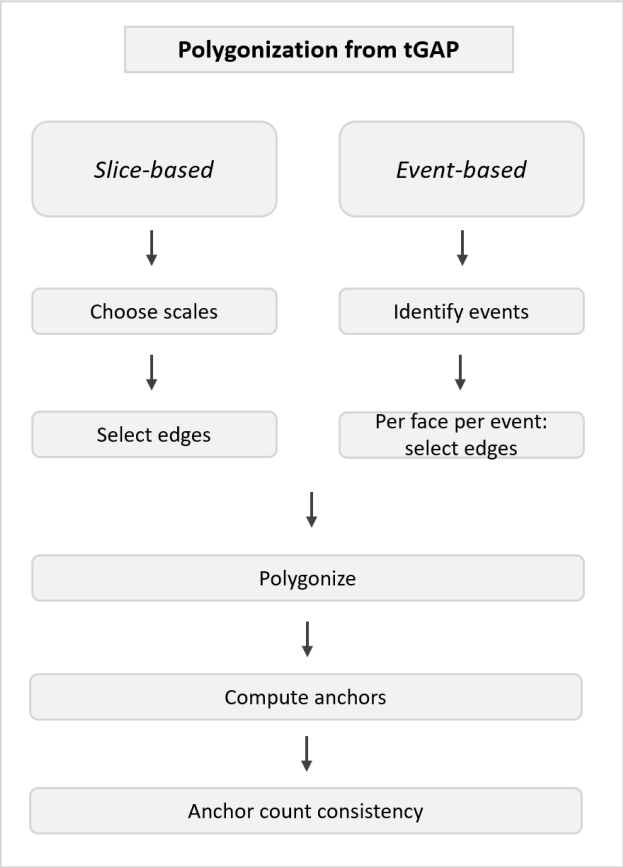


Figure 3.8.: Workflow for polygonization from tGAP

3.3.1. Slice-based Method

The slice-based method polygonizes faces at a predetermined set of map scales (discrete "slices" of the scale range) and computes anchor points on each slice. This generates a traditional multi-scale representation, e.g. choosing scales 1:10000, 1:20000, 1:40000, etc., then extracting the face geometries at each scale. By processing slices in ascending order of scale (from most detailed to most generalized), we can track and maintain consistent anchor points for each face. Table 3.2 summarizes its main steps.

We first decide on a base scale and a set of representative scales. For example, if the base (most detailed) scale is 1:10000, we then select scales doubling in denominator (1:10k, 1:20k, 1:40k, 1:80k) to cover progressively coarser representations. Each scale value is converted to the corresponding tGAP step value using the known scale-step conversion (the tGAP uses an abstract step unit; a function step_for_scale() provides the step value for a given scale denominator). The result is a list of step values $S_1, S_2, \ldots, S_n$ for which faces will be polygonized.

For each chosen step S_i, we retrieve all edges from the tGAP that are active at that step,

Step	Process	Input	Output
1	Select representative scales (slices)	tGAP dataset; base scale	Set of step values for slices $S_1 \dots S_n$
2	Retrieve edges at slice S_i	tGAP edges (with `step_low`, `step_high`)	Subset of edges active at S_i
3	Polygonize edges to faces	Edges active at S_i	Polygons representing faces at S_i
4	Attach face attributes	Polygons; tGAP face	Faces with ID, class, name
5	Compute anchor point(s)	Face polygon geometry; feature class	Anchor point(s) with orientation per face
6	Enforce anchor count consistency	Current anchors; previous anchors	Adjusted anchor list (if needed)
7	Store anchors in table	Anchor points	Cumulative anchor table (face, step, geom, angle, etc.)

Table 3.2.: Slice-based polygonization pipeline - key inputs and outputs at each stage

i.e., all edges e such that $e.step\_low \leq S_i < e.step\_high$. These edges collectively form the boundary of all faces present at that scale. We then perform a polygonization on this edge set to reconstruct the polygon geometries of faces. This could be done by a spatial SQL query using `ST_Polygonize` on the selected edges. An example query is shown below:

```
SELECT
    (ST_Dump(ST_Polygonize(e.geometry))).geom::geometry(Polygon, 28992)
    AS polygon_geom
FROM {dataset}_tgap_edge e
WHERE e.step_low <= {step_value} AND e.step_high > {step_value}
```

Each resultant polygon is associated back to its face ID and attributes by spatial containment: since the tGAP data includes a representative point (or `pip_geometry`) for each face, we join the polygon to a face record if the face's point lies within that polygon. This generates a collection of face polygons at step S_i, each with a face identifier and feature class (indicating road, water, building) inherited from the source data.

Next, we compute one or more anchor points for each reconstructed face polygon. The strategy for anchor placement depends on the feature class of the face, reflecting different geometry characteristics: for elongated or linear faces (e.g. roads, rivers) we apply a skeleton-based approach and for more compact faces (e.g. building footprints), we use a centroid or polylabel approach.

Each face thus generates either a single anchor or potentially multiple anchors (long linear features might get two or more anchor points if the skeleton has several distinct long segments). Each anchor point is stored along with its face ID, the current slice step S_i, the anchor's orientation angle, and the face's name or label text if available.

An important aspect is ensuring that anchors evolve consistently as scale changes. In the slice-based method, we process the slices in increasing order of scale (decreasing details). We maintain a record of the anchors assigned to each face at the previous (more detailed) slice. When moving to a coarser slice, if a face's geometry simplifies such that fewer anchors are needed, the algorithm naturally may produce the same or fewer anchor points. However, if a

new anchor candidate appears (for instance, two road segments merge into one), we limit the anchors to prevent the count from increasing as scale decreases. This is done by comparing the number of anchors for the face at scale S_i to the number at the last processed smaller scale S_{i-1}. If $|anchors(S_i)| > |anchors(S_{i-1})|$, the new anchors are filtered by distance to the old ones: only the anchors closest to the previous anchor positions are kept, ensuring $|anchors(S_i)| = |anchors(S_{i-1})|$. This rule preserves continuity (an anchor trace should not split into multiple anchors when zooming out). The result is a set of anchor points per face per slice, with monotonically non-increasing counts as scale becomes smaller.

3.3.2. Event-based Method

The event-based method takes a continuous view of scale and updates face reconstructions only at moments where the geometry changes. Instead of arbitrary fixed slices, we capture the geometry change events in the tGAP data, that is: the appearance or disappearance of an edge in a face's boundary, as the steps to polygonize that face. This method ensures that we compute a face's geometry (and anchor positions) at every scale where it changes. Figure 3.9 and Figure 3.10 illustrates how a face might evolve and be sampled at event points, and Table 3.3 outlines the steps of this event-driven pipeline.

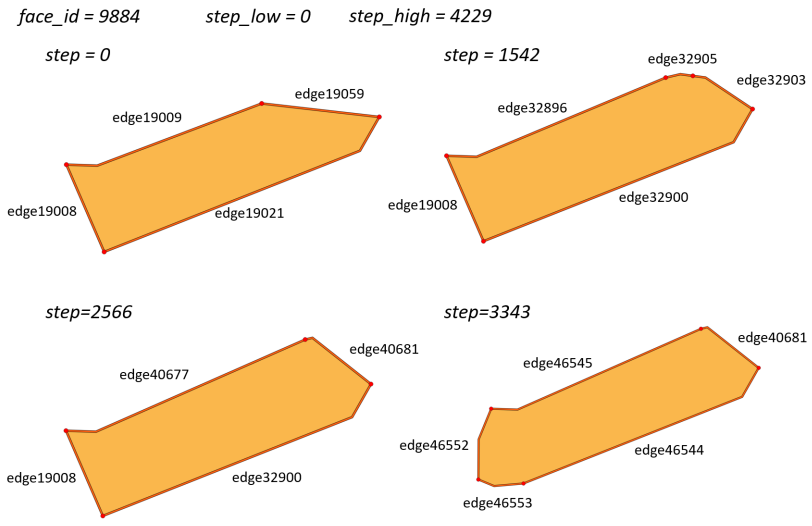


Figure 3.9.: An example of face evolution

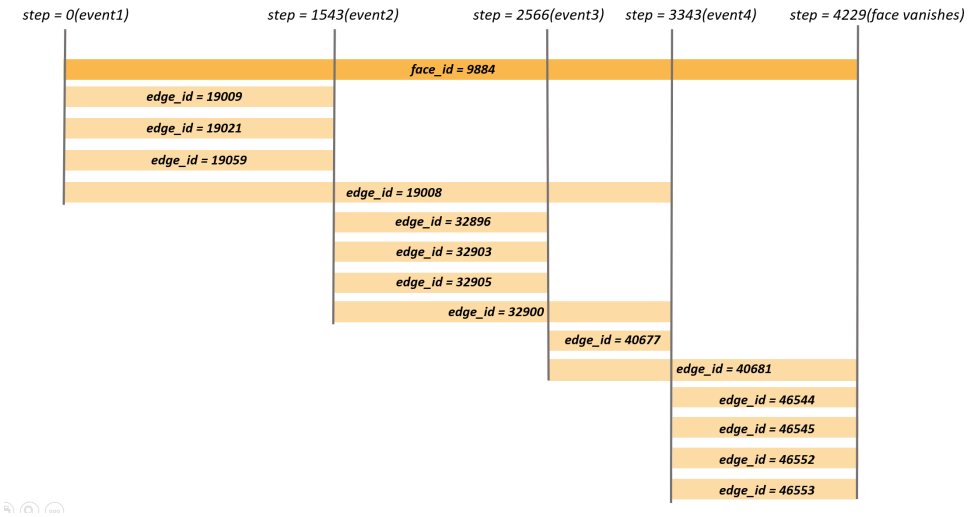


Figure 3.10.: An example of edges that bound one face at different steps and events

Step	Process	Input	Output
1	Initialize face events	Face's *step_low* and *step_high*	Initial set with face birth & end scales
2	Retrieve bounding edges for face	tGAP edges, face ID	All candidate edges for face's boundary
3	Compute edge-face overlap intervals	Face scale range; edge scale ranges	Intervals where edges bound the face
4	Augment event set with edge start/end	Edge intervals for the face	Sorted event steps $[S_1, S_2, \ldots, S_n]$
5	For adjacent event pair (S, T): select edges active at S	Edge intervals; event S	Edges forming face at scale S
6	Polygonize edges to reconstruct face	Edges active at S	Face polygon at scale S
7	Compute anchor point(s) for face	Face polygon; feature class	Anchors with orientation
8	Enforce anchor count consistency	Current anchors; previous anchors	Adjusted anchor list
9	Store anchors in table	Anchor points	Updated anchor table (face, step, geom, angle, etc.)

Table 3.3.: Event-based polygonization pipeline - key steps for reconstructing a face at its change events

We iterate through each face in the tGAP structure. For a given face (identified by `face_id`), the tGAP stores the scale range in which it exists (*face.step_low* = the largest scale at which it appears, *face.step_high* = the smallest scale at which it is still present). Within that range, the face's boundary may undergo changes as certain edges merge or vanish. We retrieve all edges that ever bound this face using a query on the tGAP edge table, any edge whose left or right face is this face will be relevant:

```
SELECT edge_id, step_low, step_high, ...
FROM {dataset}_tgap_edge
WHERE
  step_high > {face.step_low}
  AND step_low < {face.step_high}
  AND (
       left_face_id_low = {face_id}
    OR right_face_id_low = {face_id}
    OR left_face_id_high = {face_id}
    OR right_face_id_high = {face_id}
  );
```

For each such edge, we compute the interval during which the edge actually forms part of this face's boundary. This is determined by overlapping the edge's lifespan $[edge.step\_low, edge.step\_high)$ with the face's lifespan $[face.step\_low, face.step\_high)$. If there is an overlap, we record the interval

$$(start, end) = \big[\max(edge.step\_low, face.step\_low),\ \min(edge.step\_high, face.step\_high)\big)$$

These intervals mark when that edge is contributing to the face.

We then compile a set of event step values for the face: initialize it with the face's own *step_low* (birth scale) and *step_high* (end scale), and add every interval start or end value from the edges bounding the face. This set includes:

- The face's introduction scale (*step_low*)
- Its disappearance scale (*step_high*)
- Intermediate scales where boundary edges enter/leave

Sorting this set generates a sequence of scales:

$$E_1 < E_2 < \cdots < E_m$$

These mark events where the face's geometry might change. Between consecutive events E_k and E_{k+1}, the boundary composition remains stable (no edge changes), meaning the geometry is preserved. Thus, we only need to polygonize the face at *one representative scale* per interval. We choose the start of each interval for polygonization:

$$S = E_k \quad \text{for } k = 1, 2, \ldots, m-1$$

- E_1: Face introduction scale (first polygonization)
- E_m: Not used for polygonization (face vanishes at this scale)

For each event step $S = E_k$ identified for the face, we retrieve all edges that bound the face in the open interval $[E_k, E_{k+1})$. We choose edges satisfying:

$$start \leq S < end$$

, using the precomputed edge-face intervals. These edges at step S form a closed boundary around the face. We then process these edges through `ST_Polygonize` to derive the face's geometry at scale S. An example of one face's polygonization at different key scales is shown in Figure 3.11

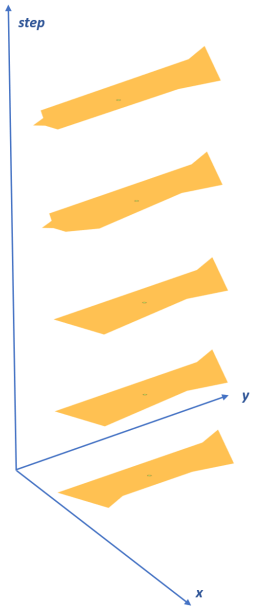


Figure 3.11.: An example of one face polygonization at different key scales

At this point we have the face geometry at the geometry changing scale S. We then compute anchor point(s) for the face exactly as in the slice-based method(Section 3.3.1): using the face's feature class to decide between the skeleton-based method (for roads, water) or the polylabel-based method (for buildings). Similar to before, as we move through the sequence of events for a face, the facs simplifies with each event, we also enforce that the number of anchors for a face does not increase at later (smaller) scales. If at a particular event E_k the anchor algorithm produces more points than the previous event E_{k-1} did, we prune the anchors to maintain the previous count. This ensures a single face's label doesn't increase when the map is zoomed out.

3.4. Label Lines Trajectory

The label lines trajectory is a construct that represents the path of a label across a sequence of discrete steps (different scales). The workflow for generating these trajectories is illustrated in Figure 3.12. The conceptual goal is to trace the same real-world feature's label through multiple steps and represent its evolution as a continuous trajectory in a space defined by x, y, *angle* and *step*. Here, x and y are spatial coordinates, *angle* is label rotation angle and *step* is an axis indicating scale level. By plotting a label's positions and rotations against the step axis, we obtain the label's trajectory through the sequence (as shown in Figure 3.13 for one face). The benefit of this representation is that it captures the label's movement or persistence over time/scale in one continuous object, rather than as disjoint states.

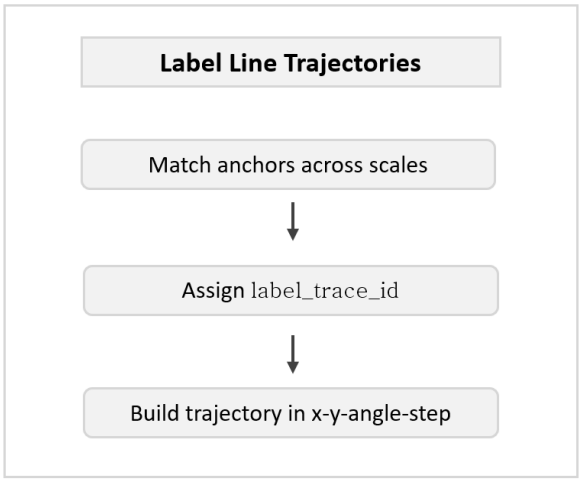


Figure 3.12.: Workflow of label line trajectory construction

Constructing label trajectories requires identifying which label anchors across different steps correspond to the same label. We introduce a persistent identifier `label_trace_id` to group label anchors that belong to the same label across steps. For long geometries, one face may have multiple label anchor points, the challenge is to assign the same `label_trace_id` to label anchors that represent the same label at consecutive steps, despite potential shifts.

For each pair of consecutive steps, we compare the sets of label anchor points and attempt to match each label in the earlier step with the nearest label in the later step (within the same face geometry). We see this problem as a matching problem, where the cost of matching two label anchors is the distance between their positions. We then solve the assignment by finding the minimum-cost matching between anchors in step *N* and step *N+1*. This generates a pairing that minimizes the total movement of labels.

Matching is constrained by feature identity: we only attempt to match labels that refer to the same underlying face(for example, the same road). This ensures we are tracking the same label rather than accidentally linking two different labels that happened to be nearby in space. The output of this process is an assignment of a `label_trace_id` to each label anchor in each step, so that all anchors with the same ID form a continuous trace across the step changes.

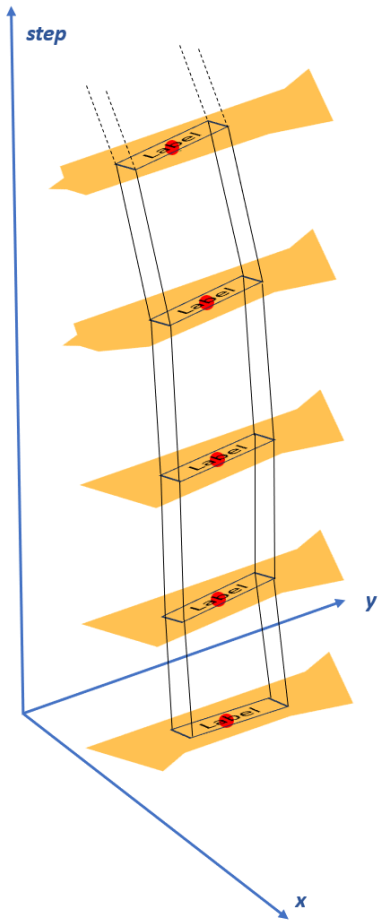


Figure 3.13.: Label line trajectory(a moving label over space)

Once persistent IDs are assigned, we can extract each trajectory as the sequence of labels (with their step values) sharing a given `label_trace_id`. By connecting these labels in order of increasing step, we form the label line trajectory through the *(x, y, angle, step)* space. These trajectories show the label's motion. Importantly, they provide a foundation for interpolation and animation: since we know a label's exact position and rotation at each discrete step, we can interpolate intermediate positions and rotations for any fractional step in between. The trajectory serves as a guideline along which the label can slide when transitions from one step to another. This smooth interpolation would not be possible if labels at each step were treated independently with no continuity.

3.5. Visualization

In this section, we integrate the previous label lines trajectory concept into an interactive visualization system that allows users to transition through steps and see labels update smoothly. The core is that each label is treated as a persistent object with a series of

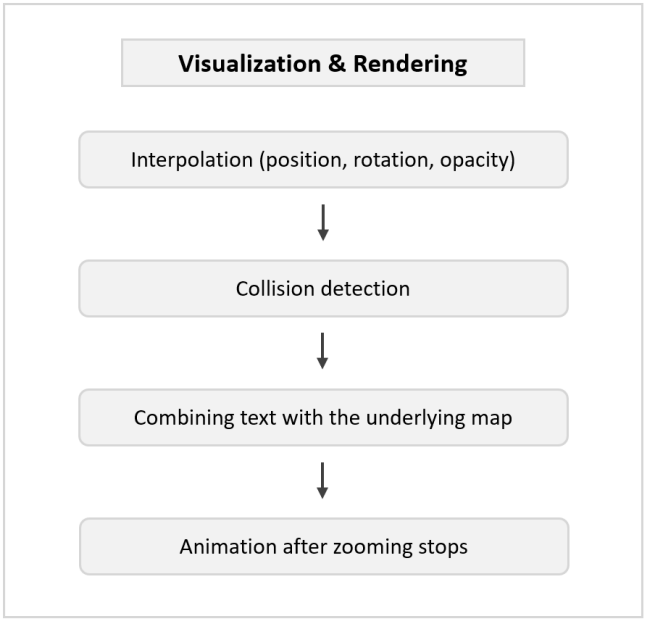


Figure 3.14.: Label visualization steps

keyframes at the known steps, the main steps are illustrated in Figure 3.14. The system stores the label's properties (such as position, orientation, text, etc.) at each step where it exists. During interaction, as the current step changes, the system computes the label's visual state at the intermediate positions using interpolation. By doing so, we avoid the jarring effect of labels "popping" in new locations; instead they move continuously to those new locations.

3.5.1. Interpolation of Labels

Given a label's position at two consecutive steps, we linearly interpolate its coordinates and rotation angle for intermediate step values(Figure 3.15).

The interpolation weight α is defined as the normalised location of the current step S within the interval delimited by its bracketing key steps. S_p is the largest key step satisfying $S_p \leq S$ (the previous key step) and S_n is the smallest key step satisfying $S_n \geq S$ (the next key step). The weight is then

$$\alpha = \frac{S - S_p}{S_n - S_p}, \qquad 0 \leq \alpha \leq 1, \tag{3.1}$$

with the degenerate case $S = S_p = S_n$ giving $\alpha = 0$ (exact key step). This scalar is used in all per-step interpolations: the label's anchor position is evaluated as

$$\mathbf{p}(S) = \mathbf{p}(S_p) + \alpha\big(\mathbf{p}(S_n) - \mathbf{p}(S_p)\big) \tag{3.2}$$

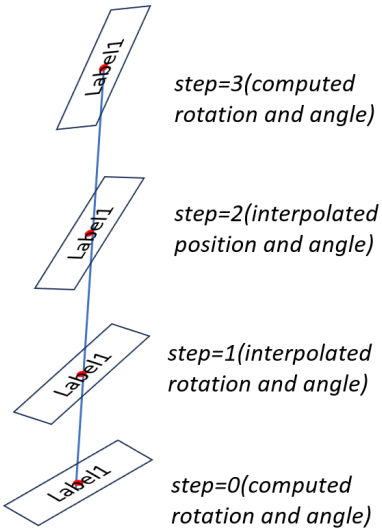


Figure 3.15.: Interpolation of label positions and rotation

Labels also have orientation to align with the underlying polygon, we similarly interpolate their rotation angles (see Figure 3.15). As a result, labels that rotate between steps will appear to turn gradually, which is visually smoother and more natural than a sudden jump in angle.

We interpolate opacity (transparency) to fade labels in or out. Each label has a visibility interval across steps. For example, a label might first appear on the map at step 0, have its last defined key step at step 3, and remain valid until step 6 (Figure 3.16). Instead of abruptly adding or removing the entire label at intermediate steps, we apply a fade.

When zooming out (i.e., as the step number increases), a label that is initially fully visible (opacity = 1, fully opaque) begins to fade out as it nears the end of its lifetime, and reaches opacity = 0 (fully transparent) when the face it labels disappears. Conversely, when zooming in (i.e., as the step number decreases), the label gradually fades in, starting from opacity = 0, and becomes fully visible (opacity = 1) by the time its key step is reached.

All these interpolations (position, rotation, opacity) are done continuously as the user interacts. The effect is that labels perform smooth transitions: they glide to new positions and gently appear or vanish.

3.5.2. Label Collision Detect

Even with correct positioning, when multiple labels are displayed simultaneously, they can overlap, leading to clutter and illegibility.

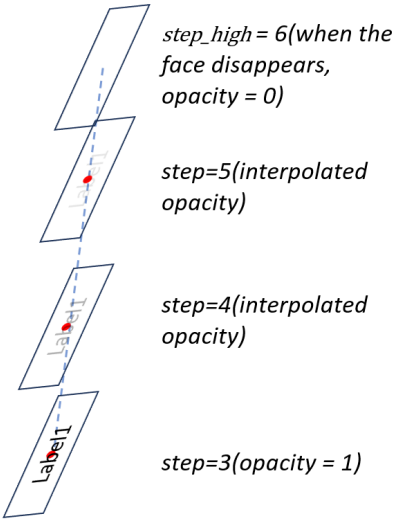


Figure 3.16.: Interpolation of label opacity

Greedy strategy with priority ranking We use a greedy strategy to handle label collisions frame-by-frame, prioritizing more important labels and hiding others as needed to prevent overlaps:

1. ranked by a *pre-computed priority* $P(\ell)$;
2. processed in descending priority order;
3. either accepted if their axis-aligned bounding-box (AABB) does not intersect any AABB already accepted
4. or discarded.

This ensures that high-priority labels are preserved, while allowing to drop conflicting low-priority labels for performance and clarity.

Priority function We define the priority of a label ℓ as the lexicographic pair:

$$P(\ell) = (step\_high(\ell), -textLength(\ell))$$

where *step_high* is the ending step of the face with which the label is attached and *textLength* is the number of characters in the label. This ensures that persistent, short labels are favored(Figure 3.17).

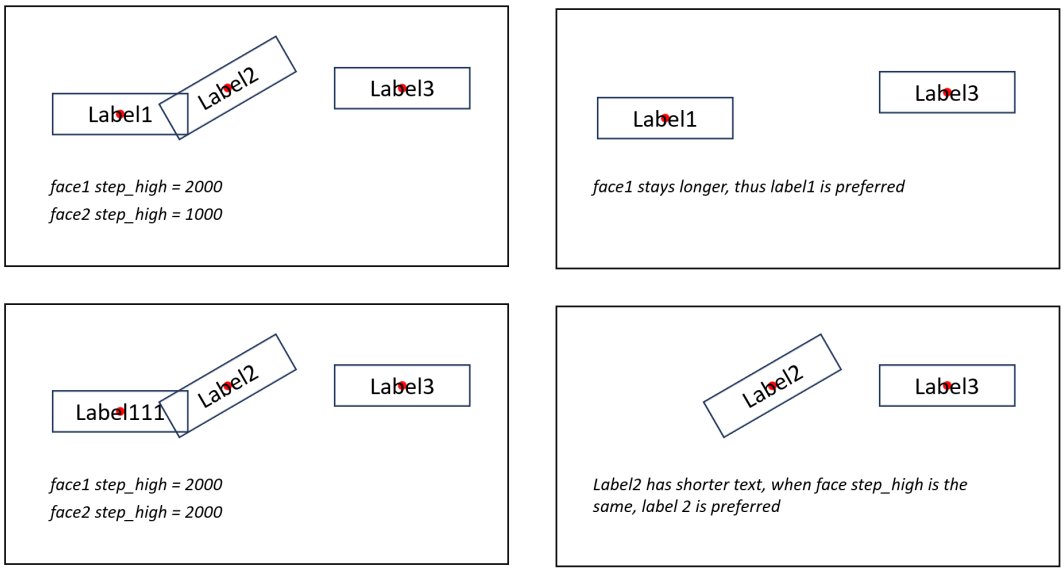


Figure 3.17.: Example of label collision detection

Computing the AABB of rotated labels We compute an *axis-aligned bounding box* (AABB) for each label, which encloses the rotated text plus padding. Given the following parameters for a label:

- Label anchor point: (x_a, y_a)
- Width: w
- Height: h
- Rotation angle: θ
- Padding: p

The process to compute the AABB is as follows:

1. Compute the padded width and height:

$$w' = w + 2p, \quad h' = h + 2p$$

2. Construct the four corner offsets from the center of the label:

$$(\pm w'/2, \pm h'/2)$$

3. Rotate each corner (x, y) around the origin (the label's center) using the 2D rotation formulas:

$$x' = x\cos(\theta) - y\sin(\theta), \quad y' = x\sin(\theta) + y\cos(\theta)$$

4. Translate each rotated corner by the anchor point (x_a, y_a) to get screen-space positions:

$$x_i'' = x_a + x_i', \quad y_i'' = y_a + y_i'$$

5. Compute the axis-aligned bounding box from the rotated corners:

$$\text{minX} = \min_i x_i'', \quad \text{maxX} = \max_i x_i'' \qquad \text{minY} = \min_i y_i'', \quad \text{maxY} = \max_i y_i''$$

Accelerated collision detection Naively checking every label against all others requires $\mathcal{O}(n^2)$ pairwise AABB comparisons(n is the number of candidate labels, i.e. all visible/interpolated labels in the current frame). To accelerate this, we use **RBush** — a library for spatial index based on a 2D R-tree[2].

RBush organizes axis-aligned bounding boxes into a balanced hierarchical structure. During rendering:

- Already-accepted label boxes are stored in the tree;
- For each new label, the tree is queried for potentially overlapping boxes using `RBush.search(box)`, which runs in expected $\mathcal{O}(\log n)$ time, where n is the number of boxes currently stored in the tree.

This reduces the number of comparisons per label from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$, assuming a reasonably balanced tree.

The revised algorithm is as follows:

Algorithm 1 Label placement algorithm

Sort L by priority descending $T \leftarrow$ empty R-tree $S \leftarrow \emptyset$ `set of accepted labels`
for $\ell \in L$ **do**
  $b \leftarrow \text{boundingBox}(\ell)$ hits $\leftarrow T.\text{search}(b)$ `O(log m) expected`
  if $\forall h \in hits : \neg overlap(b, h)$ **then**
    $S \leftarrow S \cup \{\ell\}$ $T.\text{insert}(b)$ `O(log m)`
  end
end
return S

A label is only accepted if its box is disjoint from all previously accepted boxes, but completeness is intentionally sacrificed because a lower-priority label may be hidden. This trade-off is acceptable in interactive rendering where real-time performance is important.

[2]https://github.com/mourner/rbush

3.5.3. Combining Text with the Underlying Map

To display labels in harmony with a continuously zoomable and pannable map, it is important to integrate a text rendering system that can visually align and remain synchronized with the base map rendering. This section describes how the label layer is positioned on top of the map canvas and how we ensure that the label rendering remains consistent with the underlying map.

Overlaying the label layer The map is rendered in a HTML canvas element. To add text labels without interfering with the main rendering pipeline, a second, transparent canvas is created and placed directly above the map canvas in the DOM. This overlay canvas is absolutely positioned to fully match the map viewport and styled to ignore pointer events(`pointer-events: none`), ensuring that all user interactions (e.g., drag, scroll, pinch) are directed to the map layer below.

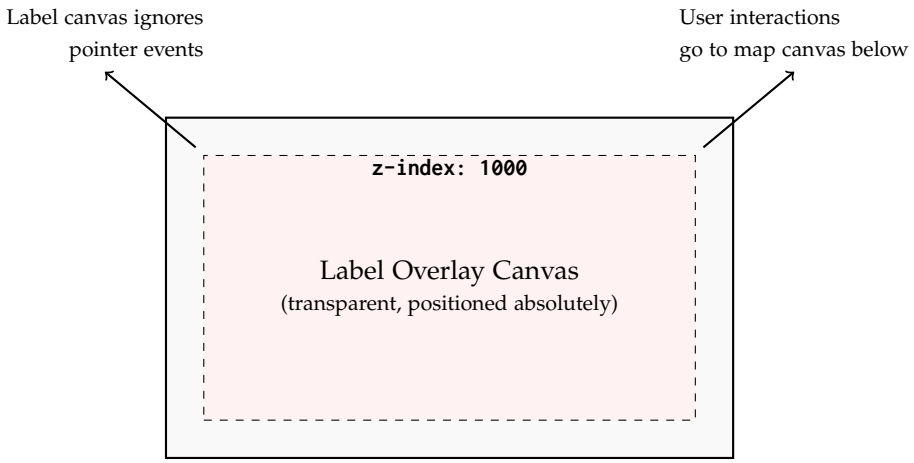


Figure 3.18.: The label overlay canvas is positioned above the map canvas using absolute positioning and a high z-index. It is transparent and ignores pointer events, allowing user interaction to reach the map layer underneath.

Synchronization of coordinate systems To ensure correct alignment, label positions must be continuously synchronized with the map's coordinate system. The map provides a transformation matrix that converts world coordinates into screen coordinates. For each animation frame, the label layer applies this matrix to transform label positions and orientations to match the current viewport.

Furthermore, the label layer listens to map events, such as viewport resizes or zoom level changes, so that it can adjust its canvas size and internal state accordingly. The label rendering system shares the same step logic and view parameters as the map, allowing both layers to remain visually consistent at all times.

Rendering Framework The technical details of how label text is rendered within this overlay canvas, including styling, opacity animation, and collision handling, are described in the following section on text rendering (Section 3.5.4).

3.5.4. Text Rendering

To render smoothly animated label text, we use PixiJS[3] — a high-performance 2D WebGL-based renderer. PixiJS was chosen due to its ability to handle thousands of display objects with smooth animations and GPU acceleration, which is important for interactive maps with frequent label updates.

Rendering hierarchy structure PixiJS uses a scene graph model, where visual elements are organized as a tree of containers and display objects. In our implementation, each label is rendered as a `PIXI.Text` object inside its own `PIXI.Container`, which is then added to a shared container for all labels. This shared container is part of the PixiJS stage. The full rendering hierarchy is illustrated in Figure 3.19.

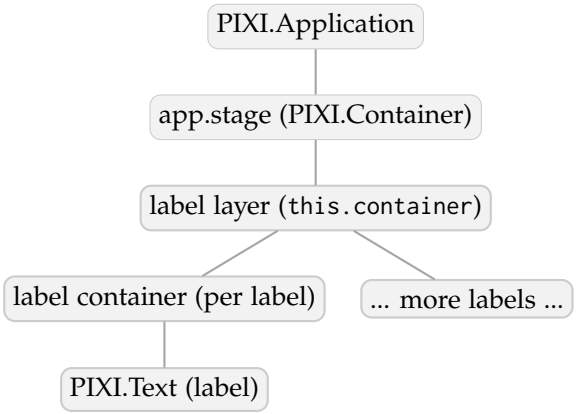


Figure 3.19.: PixiJS scene graph structure for dynamic label rendering

Overlay setup and rendering pipeline We create a transparent canvas overlay using `PIXI.Application`, which is placed directly on top of the base map canvas. This overlay is automatically resized to match the map viewport and handles all rendering operations related to labels:

```
this.app = new PIXI.Application({ backgroundAlpha: 0, ... });
parent.appendChild(this.app.view);  // Insert after the base map canvas
```

All label objects are added to a shared `PIXI.Container`, which is in turn added to the application stage.

[3] https://pixijs.com/

Label sprites and styling Each label is encapsulated as a `PIXI.Text` object and placed inside its own `PIXI.Container`. This container serves as the label's "sprite" and makes it easy to manage transformations such as position, rotation, opacity, and visibility:

```
const container = new PIXI.Container();
const txt = new PIXI.Text(label.text, style);
txt.anchor.set(0.5);  // Center anchoring for proper rotation
container.addChild(txt);
this.container.addChild(container);
```

Text styles are selected based on feature class (i.e., road, water, building), allowing for differentiated visual representations using `PIXI.TextStyle`.

Updating label states Each animation frame, label sprites are updated based on interpolated properties. The label state (position, rotation, opacity) is computed using the interpolation logic described in Section 3.5.1. World coordinates are transformed into screen coordinates using the map's world-viewport matrix.

Collision visibility The final visibility of each label is determined by the collision detection system described in Section 3.5.2. Only non-overlapping labels are set to visible:

```
if (collisions.length > 0) {
    sprite.visible = false;
} else {
    sprite.visible = true;
}
```

Rendering At the end of each frame, PixiJS renders the stage containing all visible and transformed label sprites:

```
this.app.renderer.render(this.app.stage);
```

3.5.5. Animation After Zooming Stops

A smooth label opacity animation has been implemented as in Section 3.5.1; however, labels might remain partially transparent when users stop zooming, impacting readability. To enhance the user experience, an additional animation activates once zoom interactions stop, smoothly transitioning labels to full opacity. The approach consists of three main components: zoom interaction detection, idle state transition, and frame-by-frame label opacity animation.

Detecting zoom activity Zoom activity is tracked using both mouse and touch events on the map's canvas: mouse wheel zoom is detected via the `'wheel'` event; pinch zoom (touch) is detected by tracking the distance between two fingers during `'touchstart'` and `'touchmove'` events.

Whenever zooming is detected, the following function is called to record the current time as the last zoom interaction (`lastZoomTime`) and resets the idle state:

```
const updateZoomTime = () => {
    this.lastZoomTime = Date.now();
    if (this.isZoomIdle) {
        this.isZoomIdle = false;
        this.labelOpacityOverrides.clear();
        this.zoomIdleFrames = 0;
    }
};
```

Detecting when zoom becomes idle Each frame, the code checks whether enough time has passed since the last zoom interaction. If the time exceeds a specified threshold (`zoomIdleTimeout = 500ms`), the map is considered idle:

```
    const timeSinceLastZoom = Date.now() - this.lastZoomTime;
    const isZoomIdleNow = timeSinceLastZoom > this.zoomIdleTimeout;
```

If the state changes from active to idle, the animation sequence begins.

Starting and updating opacity animation During each frame while zoom is idle, labels with opacity less than 1.0 gradually fade in:

```
if (this.labelOpacityOverrides.has(label.id)) {
    sprite.alpha = this.labelOpacityOverrides.get(label.id);

    if (this.isZoomIdle && sprite.alpha < 1.0) {
        const newOpacity = Math.min(1.0,
                            sprite.alpha + this.opacityAnimationSpeed);
        this.labelOpacityOverrides.set(label.id, newOpacity);
        sprite.alpha = newOpacity;
    }
}
```

If a label is not already animating but should fade in, it is added to the `labelOpacityOverrides` map:

```
if (this.isZoomIdle && interpolated.opacity < 1.0) {
    this.labelOpacityOverrides.set(label.id, interpolated.opacity);
}
```

The animation speed (`opacityAnimationSpeed = 0.05`) controls how fast the label fades in per frame. This results in a smooth transition that finishes in approximately 20 frames (or ~0.33 seconds) when the application ideally running at 60 FPS.

Resetting when zoom resumes If the user starts zooming again during the fade-in process, the animation is cancelled:

```
if (!this.isZoomIdle && this.labelOpacityOverrides.size > 0) {
    this.labelOpacityOverrides.clear();
}
```

All labels immediately revert to their interpolated opacity for the current zoom level:

```
sprite.alpha = interpolated.opacity;
```

This ensures that no lingering animation occurs while the zoom is active, and that the label appearance stays consistent with their logical state.

4. Results and Discussion

This chapter reports the results from applying the methodology on TOP10NL dataset in Delft region. Section 4.2 applies the proposed slice-based and event-based workflows to the dataset and makes a comparison between two methods. Section 4.3 shows the visualization results. Section 4.4 checks the outcomes against the hard and soft requirements defined earlier, highlighting both compliance and remaining trade-offs. Section 4.5 closes with a look into current limitations and the opportunities they expose for future refinement.

4.1. Research Area and Test Dataset

4.1.1. Study Area: Delft, the Netherlands

Delft offers a highly mixed cartographic landscape: canal network, irregular street pattern, residential neighborhoods, university with large footprint buildings and so on. It's an ideal ground for the testing of vario-scale labeling strategies.

All cartographic features were extracted from TOP10NL (figure 4.1), all layers are in EPSG:28992 (Amersfoort / RD New). `Wegdeel`(Road), `Waterdeel`(Water) and `Terrein`(Terrain) layers are already in planar partition in TOP10NL dataset, while adding a `Gebouw`(Building) layer needs additional data-preprocessing. Detailed scripts are provided in Appendix A. There are 12,112 faces, 25,149 edges and 15,602 nodes in total.

All cartographic features were assigned a `feature_class` code. During visualization, the following ranges are used to distinguish different feature types: values between 10,000 and 10,999 represent road features, between 12,000 and 12,999 represent water features, and between 13,000 and 13,999 represent buildings.

4.1.2. tGAP Generation Results

To prepare the data structure that underpin the slice-based and event-based labeling experiments, the pre-processed TOP10NL data were used to generate tGAP.

tGAP was generated using the existing code repository: `https://github.com/bmmeijers/tgap-ng/tree/develop`. Class compatibility matrix and class importance were not considered. The following parameters were used:

```
# whether to perform line simplification on the edge geometries
do_edge_simplification = True

% # whether to show detailed progress information
% # -- shows detailed progress on which face / edge is dealt with
```

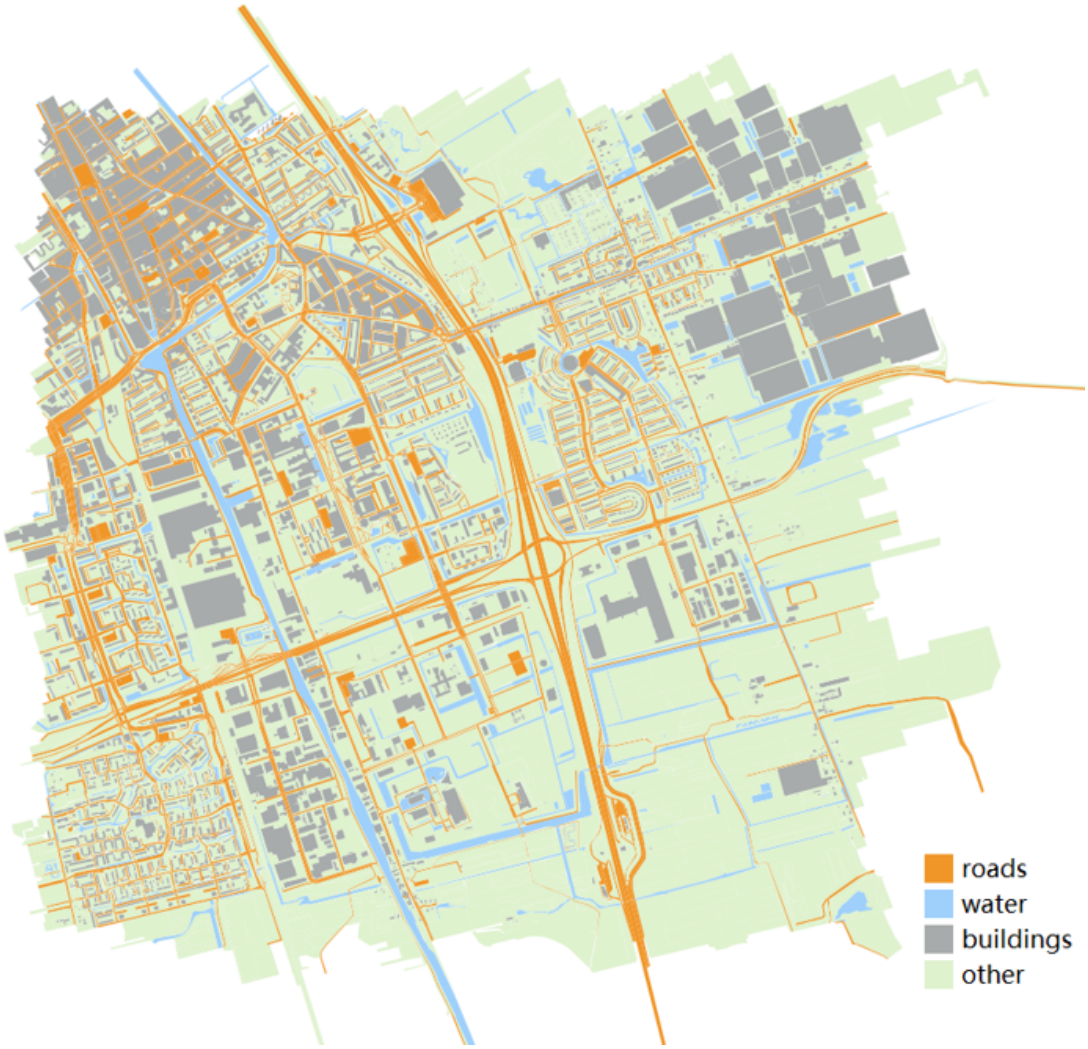


Figure 4.1.: Overview of Delft region

```
% do_show_progress = False  # rename to do_show_trace?

# whether to use the straight skeleton code as backend to generate
new boundaries while splitting areas
do_use_grassfire = True

# whether to check the edge geometries (polylines) for (self-)
intersection before and during the process
do_expensive_post_condition_check_simplify = True

# merge faces that have the same feature class
do_merge_equivalent_area_patches = False
```

The results of the tGAP generation were stored in a spatial database, details are described in Appendix B. There were 12,122 faces and 153,277 edges for the end result. A visualization of the tGAP edges and nodes is provided in Figure 4.2.

Figure 4.2.: Visualization of tGAP edges and nodes

4.2. Label Placement Results of Two Different Methods

4.2.1. Slice-based Method Result

Figure 4.3.: Map slices at scales 1:10 000, 1:20 000, 1:40 000 and 1:80 000

Following the procedure in Section 3.3.1, the base scale was set to 1:10 000 and three additional slices were created by doubling the scale denominator, i.e. at 1:20 000, 1:40 000 and 1:80 000 as shown in Figure 4.3.

For each slice, we generated (i) a skeleton geometry and (ii) a set of label anchor points for the following feature types: `Wegdeel` (road), `Waterdeel` (water), and `Gebouw` (building).

Figure 4.4, Figure 4.5, Figure 4.6, and Figure 4.7 present the main skeleton geometries for each slice. The corresponding label anchor points are shown in Figure 4.8, Figure 4.9, Figure 4.10, and Figure 4.11.

Every anchor point is stored in the database with attributes:

```
(label_id  INTEGER PRIMARY KEY,
 face_id   INTEGER,
 step_value INTEGER,
 feature_class INTEGER,
 name        TEXT,
 anchor_geom GEOMETRY(Point, 28992),
 angle       DOUBLE PRECISION,
 label_trace_id INTEGER)
```

Table 4.1 summarizes the number of label anchor points across different slices. Starting from 10,697 anchors at the base scale of 1:10, 000, the number sharply declines with each successive slice, reaching only 51 anchors at 1:40, 000. This represents a cumulative reduction of approximately −99.5%. As the map scale decreases, only the most essential features are retained, thus the most essential labels would be remained.

Table 4.1.: Slice statistics

Slice scale	`step_value`	Anchor points	Reduction vs. previous slice
1:10 000	0	10 697	—
1:20 000	9 084	2 008	−81.2%
1:30 000	11 355	218	−89.1%
1:40 000	11 923	51	−76.6%

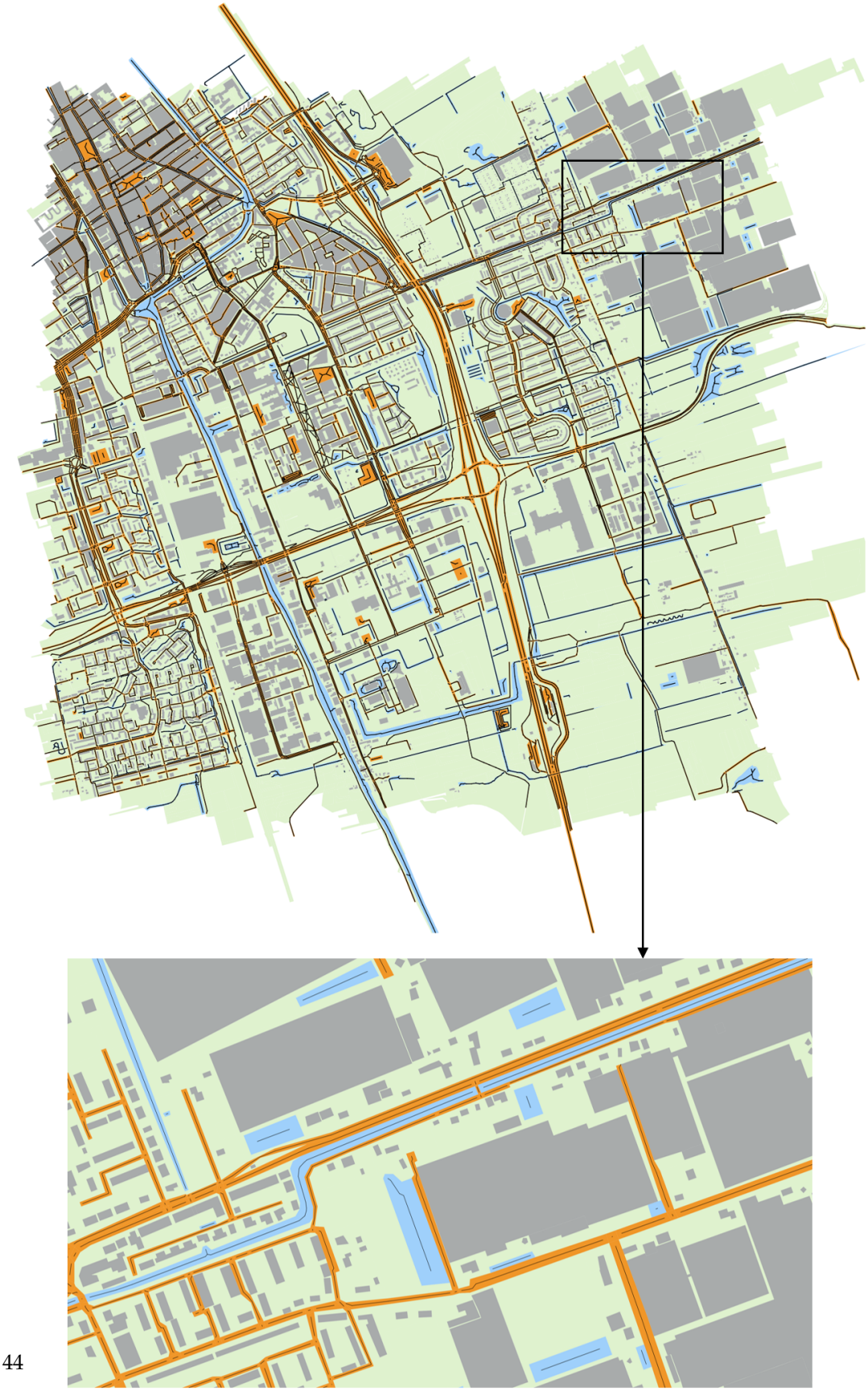


Figure 4.4.: Skeleton of map slice at scale 1:10 000

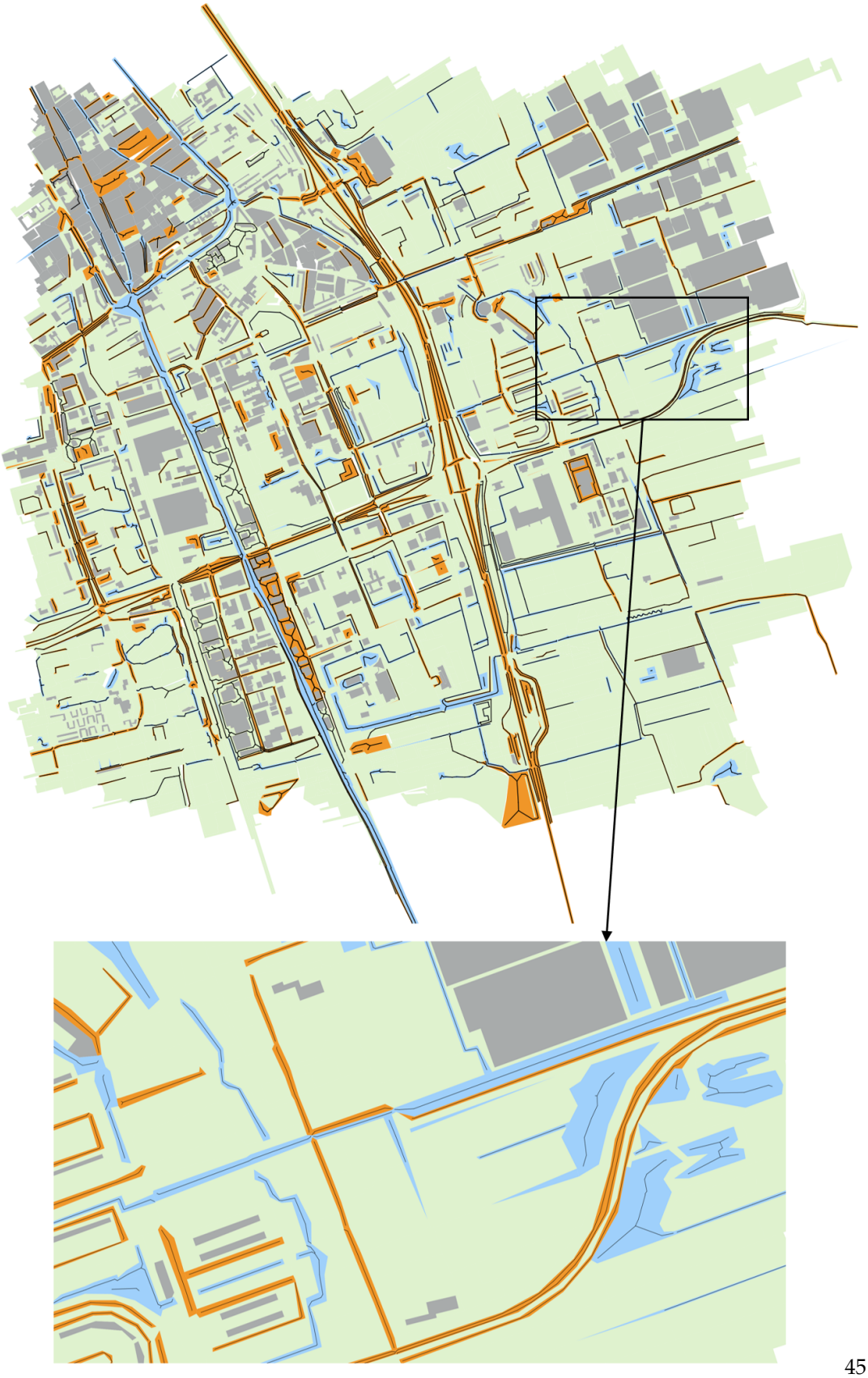


Figure 4.5.: Skeleton of map slice at scale 1:20 000

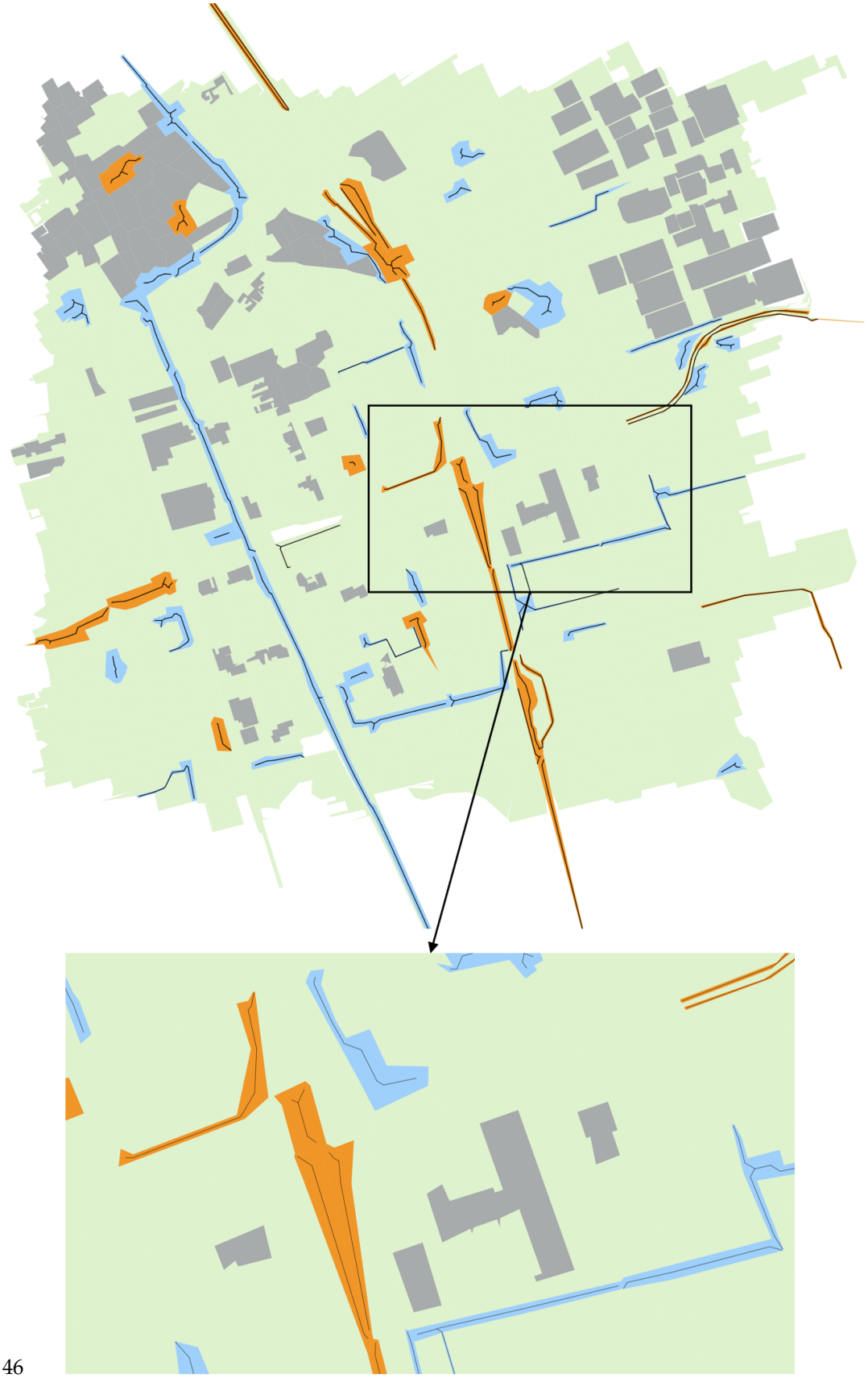


Figure 4.6.: Skeleton of map slice at scale 1:40 000

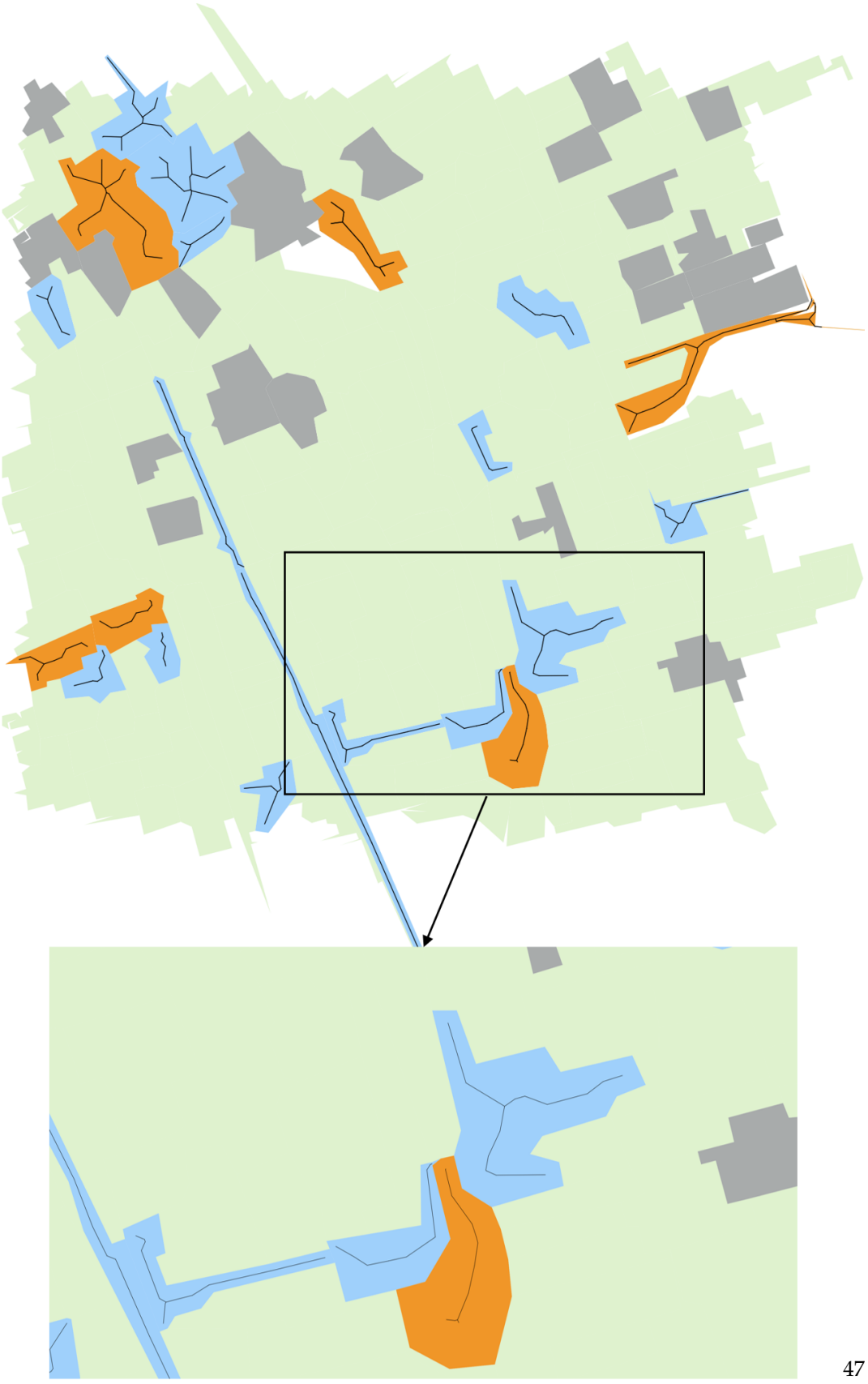


Figure 4.7.: Skeleton of map slice at scale 1:80 000

Figure 4.8.: Label anchor points of map slice at scale 1:10 000

Figure 4.9.: Label anchor points of map slice at scale 1:20 000

Figure 4.10.: Label anchor points of map slice at scale 1:40 000

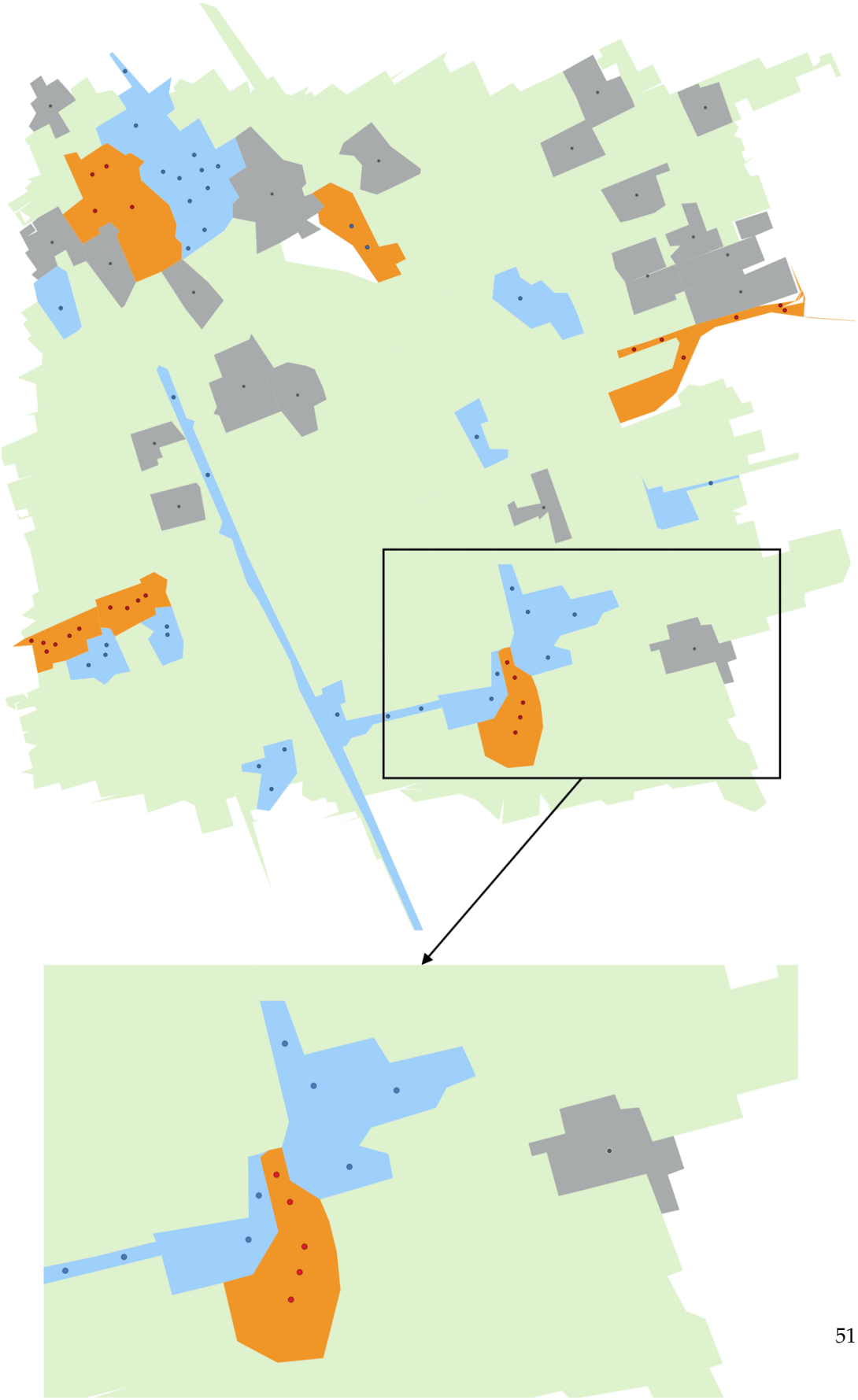


Figure 4.11.: Label anchor points of map slice at scale 1:80 000

4.2.2. Event-based Method Result

Every time the geometry of a face changes, the event-based algorithm recomputes the corresponding label anchor points (Section 3.3.2).

To quantify how often this operation is executed over the complete scale transition, all key-frame events were counted per `face_id`. Figure 4.12 shows 50 faces with most event changes.

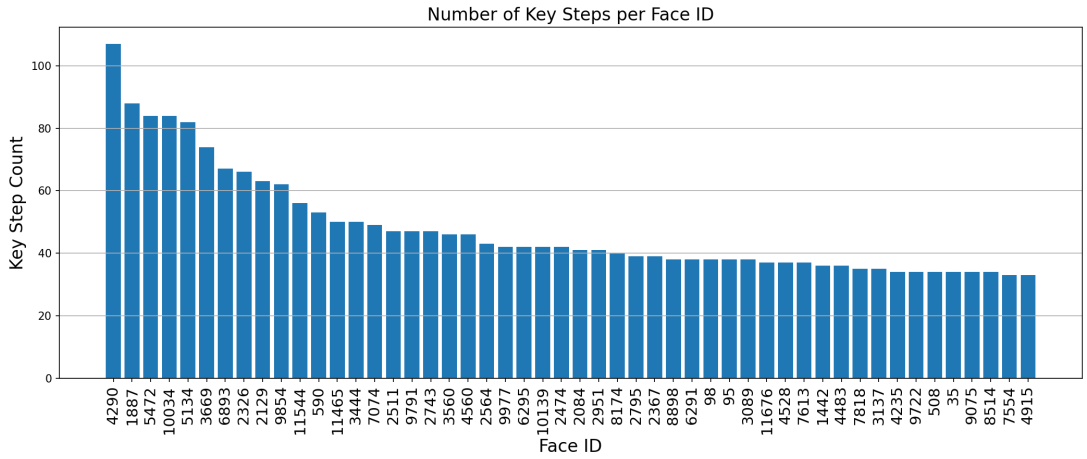


Figure 4.12.: Number of geometry change events per face (top 50)

While geometry changes can be quite subtle on individual faces over small steps, Figure 4.13 illustrates several distinct geometry changes for face `4290`. As shown, this water area progressively expands over time, as indicated by the increasing `step_value`.

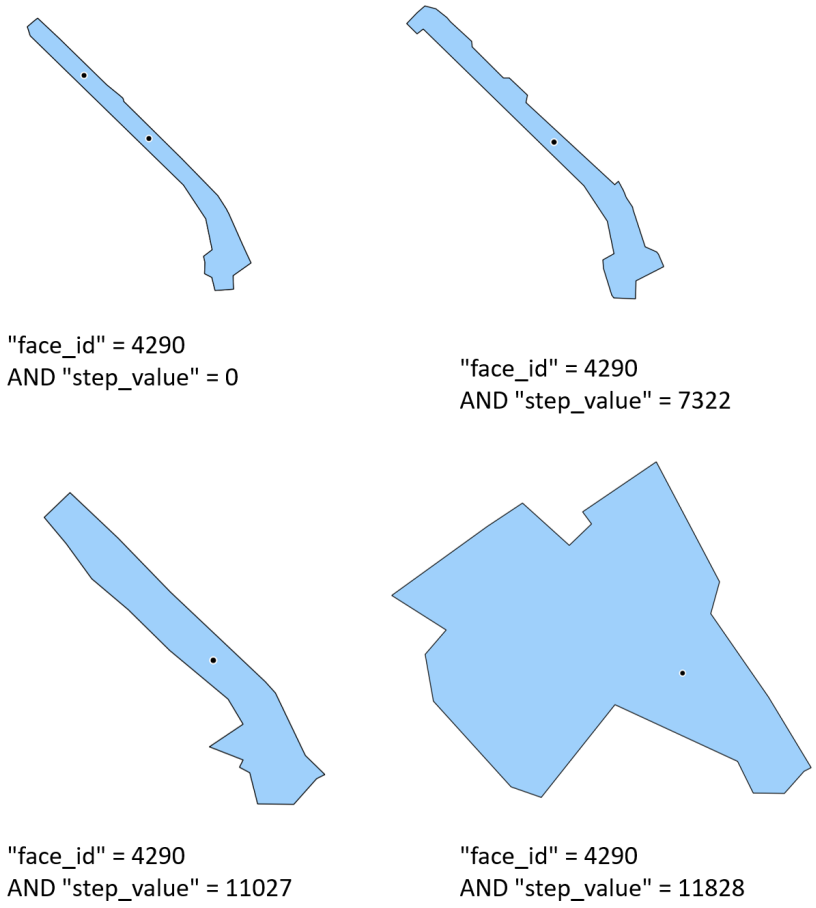


Figure 4.13.: Geometry changes of face `4290`

4.2.3. Comparison Between Two Methods

Anchor Displacement per Consecutive Scale Step

To quantify how smoothly labels transition during zooming, we measured the displacement of anchor points between consecutive steps for both labeling methods.

Smoothness was quantified as the Euclidean distance a label moves between *every* integer `step_value` in the tGAP sequence. Since key steps in both methods may be sparse, anchor positions were densified via linear interpolation to obtain positions at every intermediate scale step. We then computed the step-to-step displacement and summarized the results statistically per method. Table 4.2 reports the main statistics.

Since we interpolate label positions at every integer step, the majority of steps involve little movement in real-world coordinates. As a result, the mean displacements for both methods are extremely low, collapsing to 2 millimeter or less.

Table 4.2.: Label anchor smoothness statistics

Method	Mean	p99	Max
Event-based	0.002	0.026	**216.715**
Slice-based	0.002	0.025	0.506

However, notable differences appear when we examine the 99 percentiles and maximum values. In the even-based method, the 99th percentile displacement remains modest (2.6 cm), indicating smooth transitions for the majority of labels. Nevertheless, significant outliers exist, such as a maximum displacement of 216.715 meters. These large jumps occur when a geometry event (such as a face merge or split) triggers a complete recomputation of the label anchor point.

Figure 4.14.: Anchor displacement due to geometry change in one face

For example, Figure 4.14 illustrates a single-step jump in which a minor difference to the

edges of a road face triggers a major change in the skeleton structure, and thus results in a drastic shift in the anchor position between step 12048 and step 12049. However, although the distance between the anchor points is 216.715 meters in the real world, this translates to around 5.91 pixels on screen at step 12048 based on a screen resolution of 96 PPI and the corresponding map scale at that step.

In contrast, the slice-based method has a much tighter control over per-step movement. Since label positions are linearly interpolated between fixed key steps (i.e., predefined scale slices), the interpolation guarantees a hard upper bound on step-to-step movement, never exceeding 0.506m in the experiment. This decreases abrupt relocations but introduces gradual drift.

This difference stems from the fundamental design of the two methods:

Event-based method recomputes anchor points only when the geometry changes. Most of the time, labels have spatial stability; however, when a geometry event does occur, the recalculated anchor may differ significantly, resulting in abrupt jumps.

Slice-based method anchors are defined only at discrete scale levels (e.g., 1:10,000, 1:20,000, etc.) and interpolated linearly across intermediate steps. This ensures continuity, but the label is forced to move gradually even if the underlying geometry remains unchanged.

Total Number of Label Anchors Generated

Beyond smoothness, an important practical consideration is the number of anchor points each method generates, and consequently, the amount of data that must be stored across all scales.

The slice-based method, using four discrete zoom levels, produces at most four anchor positions per label (one per slice), assuming the label persists across all levels. In the Delft dataset, this resulted in a total of **12,974** anchor records for all area labels. In contrast, the event-based method generates a new anchor each time a face's geometry changes. This leads to a higher number of anchors per label when multiple events occur during a label's lifespan. As a result, the event-based method produced **49,018** anchor records in total.

This increase reflects the finer granularity of the event-based approach: anchors are updated only when meaningful changes occur, providing high positional precision. By contrast, the slice-based method trades off storage efficiency against positional precision, interpolating between fewer and fixed anchor positions.

From a data storage perspective, the slice-based method is more compact and may be preferable in memory-constrained environments. Meanwhile, the event-based method, though more data-intensive, maintains a tighter alignment with geometric changes and supports more accurate label positioning overall.

Computational Complexity

We now consider the algorithmic complexity of the two labeling methods.

The slice-based method has a relatively straightforward workflow: it computes optimal label anchor positions at a limited number of fixed scale slices (four in this study), and then applies linear interpolation to derive positions at intermediate zoom levels. The computational

complexity grows approximately with the number of labels L and the number of slice levels S, resulting in a total complexity of roughly $\mathcal{O}(L \cdot S)$.

The event-based method is more sophisticated. It monitors the vario-scale structure for changes and recomputes anchor positions and label rotations whenever a geometry change event is triggered. In the worst-case scenario where every label responds to every event, the complexity can approach $\mathcal{O}(L \cdot E)$, where E is the total number of events. In practice, however, most labels are only affected by a subset of relevant events, resulting in lower actual workload. Still, in the Delft dataset, the event-based method involved more anchor computations than the slice-based method, since $E > S$.

4.3. Label Visualization Result

4.3.1. Interpolation of position, orientation and opacity

Since dynamic media cannot be embedded in this thesis, only static screenshots are provided; the interactive demo is available online at:
https://imyangao.github.io/Vario-scaleMapLabelingDemo/.

There is only subtle difference between the slice-based and event-based methods in terms of visualization results for this dataset. Here, we present the results using the event-based method. Figure 4.15 to 4.18 illustrate the result of linear interpolation between key steps for label positions. Different feature types of geometries use different font colors for their labels.

Each frame corresponds to a different intermediate step and demonstrates how labels glide smoothly between anchor points. Without interpolation, labels would abruptly "jump" between positions, resulting in visual discontinuities. By contrast, interpolation enables continuous transitions, improving the visual experience during zooming.

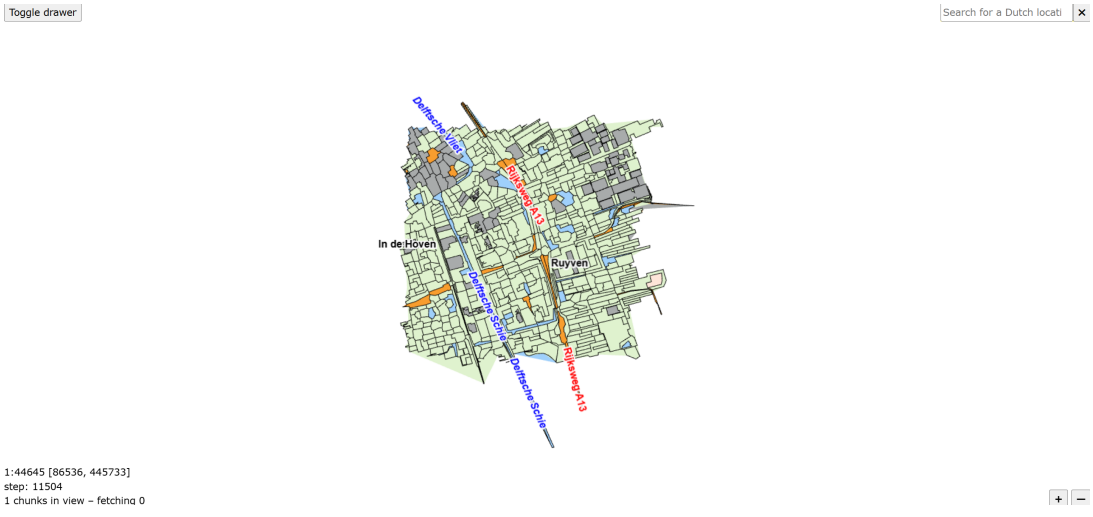


Figure 4.15.: Label display at scale 1:44645

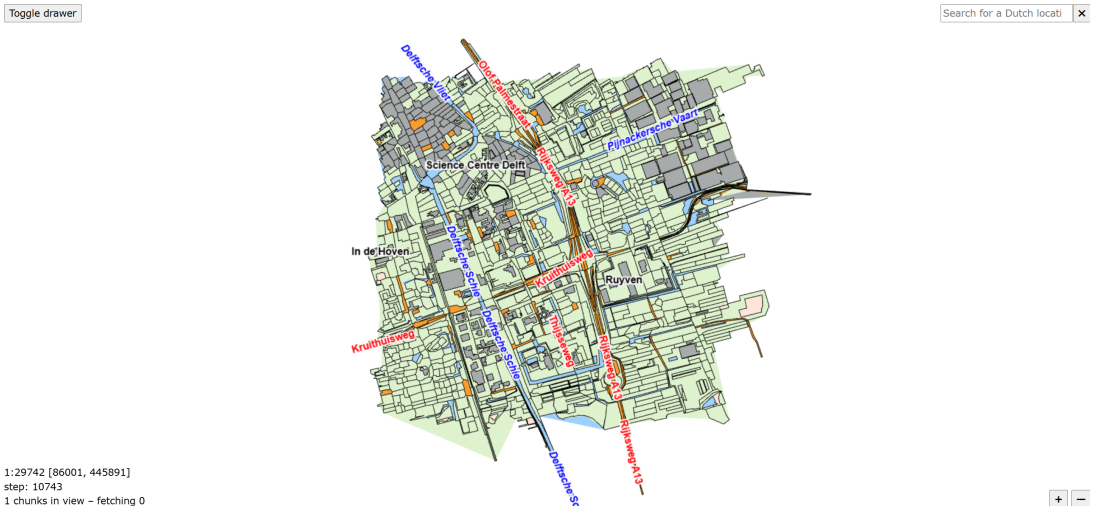


Figure 4.16.: Label display at scale 1:29742

Figure 4.17.: Label display at scale 1:21139

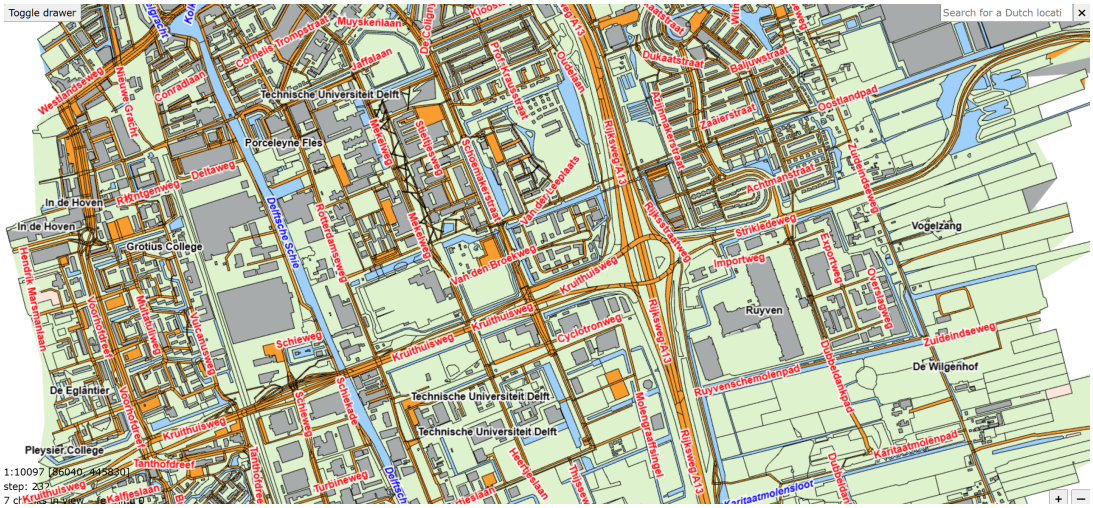


Figure 4.18.: Label display at scale 1:10097

In addition to positional smoothness, label rotation plays a role in readability, especially for features such as curved roads and rivers, where alignment improves legibility. Smooth angular transitions preserve the label orientation, avoiding abrupt changes that can distract the reader. In contrast, building labels are displayed horizontally to ensure a clear presentation. When zooming stops, all labels are rendered fully opaque; no partially transparent states are displayed.

4.3.2. Label Collision and Visibility

Greedy collision detection optimized with an R-tree structure was implemented to manage label overlaps efficiently. Figure 4.19 and Figure 4.20 compare the results of a naive approach (without any collision handling) and the optimized greedy approach. The optimized method preserves label clarity by displaying only high-priority labels while suppressing lower-priority ones that would otherwise overlap.

The greedy collision detection ensures that the map remains legible and uncluttered during scale transitions, maintaining interactive performance without sacrificing real-time responsiveness.

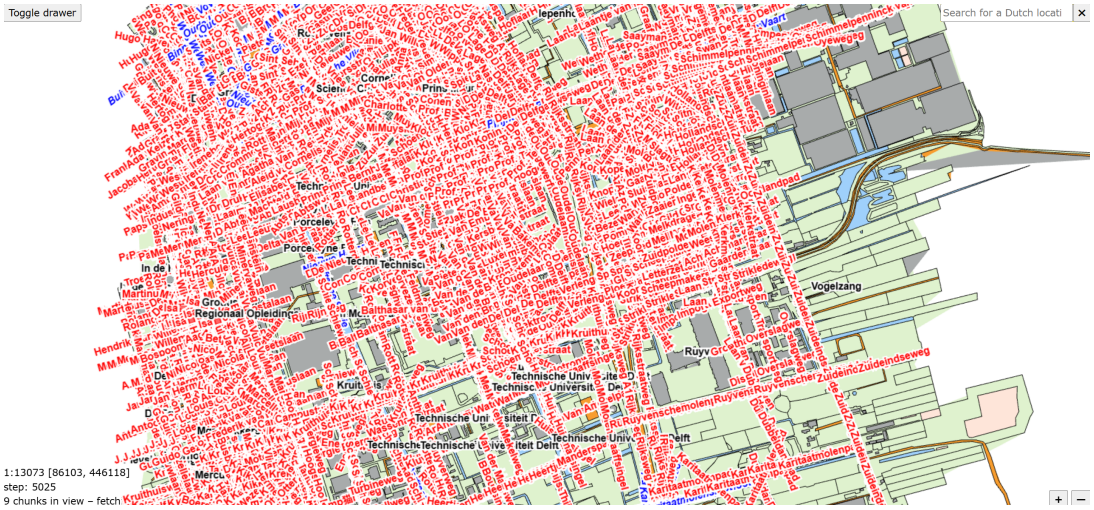


Figure 4.19.: Labels without collision detection

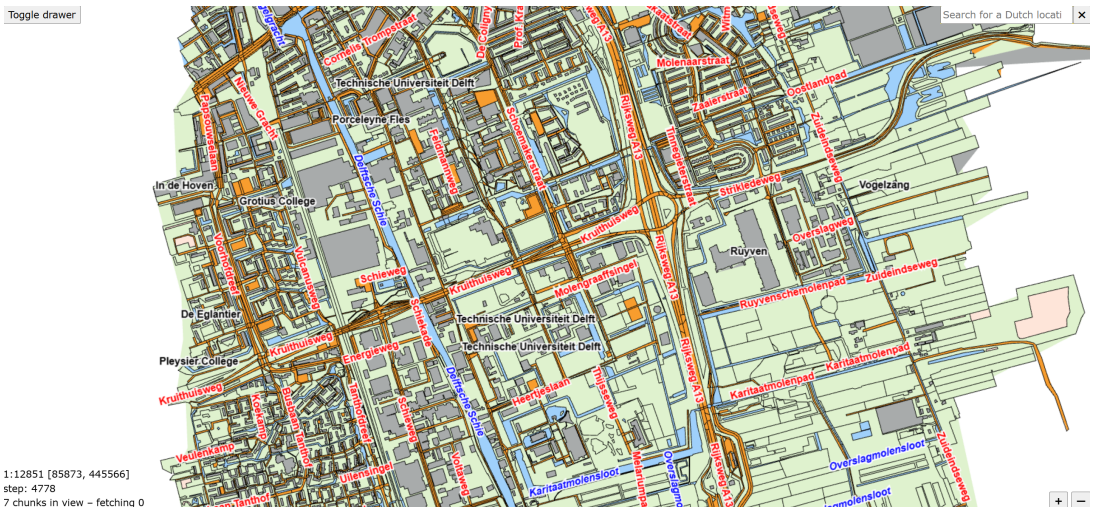


Figure 4.20.: Labels with collision detection

4.4. Evaluation of Label Placement Requirements

This section evaluates the implemented label placement methodology against the previously defined label placement requirements (see Section 3.1). The evaluation addresses both the fulfillment of the hard requirements and explains why certain soft requirements were not implemented.

4.4.1. Evaluation of Hard Requirements

The implemented approach fully meets four hard (must-have) requirements but failed one:

H1. Collision-free rendering: the greedy collision detection algorithm combined with R-tree acceleration ensures no labels overlap at any visible scale. This is demonstrated in Figure 4.20, where all labels are rendered without collisions.

H2. Unambiguous feature association: each label is explicitly linked to a single feature through precise anchor placement methods.

H3. Label continuity during continuous zoom: the interpolation-based method provides smooth transitions of label position, rotation, and opacity between discrete scale steps. Figures 4.15 through 4.18 illustrate the gradual movement of labels between key steps.

H4. Deterministic priority model: labels are prioritized using a deterministic rule based on the `step_high` value and text length, this ensures consistent label selection under identical input conditions.

H5. No flicker: when collision detection is performed on a per-frame basis, labels that appear in one frame may be removed in the next, leading to an undesirable flickering effect.

4.4.2. Discussion on Soft Requirements

Some soft requirements were not implemented, either intentionally or due to practical constraints:

S1. Avoid half-transparent rest state: to reduce confusion caused by half-transparent labels, an animation is applied once zooming stops, ensuring that labels reach full opacity in their final state.

S2. Micro-adjust before drop: the idea of fine-tuning label positions to prevent removal was omitted. Although this could increase label density, it adds large computational overhead and could negatively affect real-time performance.

S3. Preserve horizontal orientation: to improve readability, building labels are deliberately rendered with a fixed horizontal orientation.

4.5. Limitations

Several limitations inherent to the proposed approach must be considered when applying the methodology in practice. These limitations include both algorithmic constraints and practical implementation challenges:

1. Quality of label anchor points: anchor points computed using skeletonization method might not always generate optimal results. Skeletonization, particularly in geometries with complex or irregular boundaries, can produce excessive branching or misplaced midpoints.

2. Linear interpolation only: interpolation between discrete keyframes is strictly linear, providing smooth transitions but may be potentially less visually appealing or realistic compared to non-linear interpolation methods such as easing curves. Linear interpolation may

result in labels moving at uniform speed, which can sometimes feel mechanical to users interacting with the visualization.

3. Interaction between interpolation and collision detection: although interpolation ensures smooth positional transitions, the collision detection mechanism may override these smooth transitions with abrupt visibility changes. For instance, a lower-priority label may suddenly disappear when overlapping, or a label may abruptly appear when space becomes available. These sudden changes partially undermine the intended continuity of the zoom interaction.

5. Conclusion

5.1. Research Overview

This research aimed at developing label placement methods specifically designed for vario-scale maps, focusing on continuous label transitions and smooth adjustments across scales. The core research question addressed **how can labels be dynamically placed and adjusted on vario-scale maps to maintain readability, usability, and visual coherence across continuously changing scales**.

To provide a comprehensive answer, several sub-questions guided this study:

1. **Label placement requirements: what are the hard and soft requirements for optimal vario-scale label placement?**

   The hard requirements, such as collision-free rendering, unambiguous feature association, label continuity during zooming, deterministic behavior, and avoidance of flicker, were treated as non-negotiable constraints during implementation. These were essential to maintain the map's readability across scales.

   Soft requirements, including strategies like avoidance of half-transparency state, micro-adjustment before label removal, and preservation of horizontal orientation, were considered desirable but optional. They were implemented where feasible to improve cartographic quality without compromising the hard constraints.

2. **Optimal placement techniques: how can the optimal positions for labels be determined for both elongated and more compact features, ensuring spatial alignment and visual clarity?**

   To ensure spatial alignment and visual clarity, this thesis adopts feature-specific placement strategies for optimal label positioning. For elongated geometries such as rivers or roads, skeletonization was applied to derive central paths along which multiple labels could be placed. Label anchors were positioned at midpoints of long skeleton segments, with orientation aligned to the segment's main direction. This approach ensures distributed, aligned labeling across the feature's extent.

   For compact geometries such as buildings, a single anchor point was used to avoid clutter. The default anchor was the centroid, offering a stable and central location. When the centroid fell outside the polygon (e.g., for concave shapes), the pole of inaccessibility was used as a fallback. Label orientation was derived from the major axis of the feature's minimum rotated bounding box to preserve visual alignment.

3. **Dynamic adjustments: how can labels be smoothly transitioned across scales, minimizing positional shifts during scale changes, while maintaining readability and visual continuity?**

In this research, two polygonization strategies(slice-based and event-based) were developed and assessed for deriving polygon geometries from the tGAP structure to facilitate dynamic label placement. The slice-based method involved reconstructing polygon geometries at predetermined, discrete scales; the event-based method captured polygon geometries precisely at scales where geometry changes occurred.

To ensure smooth transitions across scales, label anchors were computed at discrete steps based on two different polygonization strategies, and subsequently interpolated to intermediate scales. Label anchors were linked into continuous trajectories through a nearest-neighbor matching approach, minimizing positional shifts. Linear interpolation was initially used between discrete scales, ensuring labels transitioned smoothly without abrupt jumps, enhancing visual continuity and readability.

4. **Data structure and retrieval: how can label-related data be structured and stored efficiently to support dynamic rendering?**

   Label metadata was encoded based on the tGAP structure, associating relevant geographic features on the map with label information at discrete scale steps. Specifically, the data structure records anchor point coordinates, label orientation angles, name text, and unique trace identifiers (trace IDs) for each feature at predefined key steps. This encoding supports label rendering during zoom: when the view changes scale, the system queries the corresponding scale of the vario-scale maps and displays each label at its precomputed anchor.

5. **Dynamic display: how can label text be effectively placed on maps in a way that integrates with the underlying geographic features and supports continuous scale changes?**

   A dedicated overlay canvas is positioned above the base map and synchronized with its coordinate system, allowing label positions to remain consistent during panning and zooming. Labels are rendered using PixiJS, a framework that supports smooth animations, hierarchical scene management, and per-label transformations.

   Label placement and orientation are updated in each animation frame based on world-to-screen transformations, ensuring alignment with map features. Interpolation techniques maintain continuity during scale transitions, while collision handling ensures labels remain non-overlapping.

In conclusion, this research developed and evaluated a methodology for dynamically placing and managing labels on vario-scale maps. It introduced a requirement framework distinguishing between hard and soft constraints, applied different anchor computation strategies for elongated and compact geometries to determine suitable label positions, and ensured smooth transitions across scales using interpolation of anchor trajectories. Furthermore, it structured the data to encode label information and implemented a synchronized overlay rendering system to maintain alignment with the base map. Together, these components support continuous and readable label placement integrated with the vario-scale maps.

5.2. Contributions

This study has provided several key contributions to the field of labeling vario-scale maps:

1. **Novel integration method**: new method was introduced for combining dynamic label transitions with the vario-scale tGAP data structure, supporting label continuity and smoothness during scale transitions.

2. **Anchor-based trajectory approach**: the concept of persistent label trajectories derived from anchor points was implemented, improving label continuity and visual coherence.

3. **Practical visualization approach**: developed an interactive visualization solution that demonstrated real-time smooth transitions and effective collision avoidance. The system is integrated with the underlying vario-scale base maps, ensuring alignment during panning and zooming. The approach was validated through application to the TOP10NL dataset of Delft.

These contributions extend current understanding and techniques, providing a basis for future development in labeling vario-scale maps.

5.3. Future Work

Although this study made some progress in labeling vario-scale maps, several aspects still remain for future exploration to refine and extend the methodology:

1. **3D skeletonization:** currently, skeletons are computed in 2D per scale step, which can lead to discontinuities in anchor placement when transitioning across scales. A possible direction is to investigate 3D skeletonization techniques that operate directly in the space-scale cube. By treating the evolving geometry as a 3D volume (with scale as the third dimension), a continuous 3D skeleton could be extracted, potentially generating more stable and coherent anchor paths for dynamic label placement.

2. **Interpolation methods:** rather than relying solely on linear interpolation, future work could explore the use of easing functions (e.g., Bézier curves) to produce more natural label movements across scales. These techniques may help preserve spatial continuity better.

3. **Dynamic data transmission:** to improve performance in web-based environments, future systems could adopt progressive loading strategies, delivering label data incrementally based on user view and zoom level. This could reduce initial load time and support scalability to large datasets.

4. **Collision management:** current per-frame collision checks can cause labels to flicker due to sudden visibility toggling. Future work could reduce this effect by applying opacity fading during appearance/disappearance or introducing some delay in the collision logic so that a label is only removed after sustained overlap (e.g., over several frames), rather than immediately.

5. **Integration into generalization processes:** an interesting direction is to explore whether label content (e.g., place names) can inform generalization decisions in the tGAP construction, such as deciding which features to merge or preserve. Incorporating label relevance or semantic weight into the generalization pipeline may improve both the resulting map geometry and labeling quality.

6. **Support for additional interactions and features:** extending the system to support interactive map rotation or multi-language labels would enhance its utility. For multilingual labeling, precomputing label trajectories for each language and dynamically swapping them based on user preference could provide a seamless experience.

Bibliography

V. Agafonkin. polylabel: A fast algorithm for finding the pole of inaccessibility of a polygon. https://github.com/mapbox/polylabel, 2024.

M. Barrault. A methodology for placement and evaluation of area map labels. *Computers, Environment and Urban Systems*, 25(1):33–52, 2001. doi: 10.1016/s0198-9715(00)00039-9.

M. Barrault and F. Lecordix. An automated system for linear feature name placement which complies with cartographic quality criteria. In *AUTOCARTO-CONFERENCE-*, pages 321–330, 1995.

K. Been, E. Daiches, and C. Yap. Dynamic map labeling. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):773–780, 2006. doi: 10.1109/tvcg.2006.136.

K. Been, M. Nöllenburg, S.-H. Poon, and A. Wolff. Optimizing active ranges for consistent dynamic map labeling. *Computational Geometry*, 43(3):312–328, 2010. ISSN 0925-7721. doi: https://doi.org/10.1016/j.comgeo.2009.03.006. URL https://www.sciencedirect.com/science/article/pii/S0925772109000649. Special Issue on 24th Annual Symposium on Computational Geometry (SoCG'08).

S. Biniek, G. Touya, and G. Rouffineau. Fifty shades of roboto: Text design choices and categories in multi-scale maps. *Advances in Cartography and GIScience of the ICA*, 1:1–8, 2019. doi: 10.5194/ica-adv-1-2-2019.

D. Dörschlag, I. Petzold, and L. Plümer. Placing objects automatically in areas of maps. 01 2003.

S. Edmondson, J. Christensen, J. Marks, and S. Shieber. A general cartographic labelling algorithm. *Cartographica*, 33(4):13–24, 1996. doi: 10.3138/U3N2-6363-130N-H870. URL https://doi.org/10.3138/U3N2-6363-130N-H870.

O. Ertz, M. Laurent, D. Rappo, A. Sae-Tang, and E. Taillard. Pal-a cartographic labelling library. 01 2008.

D. Garcia-Castellanos and U. Lombardo. Poles of inaccessibility: A calculation algorithm for the remotest places on earth. *Scottish Geographical Journal*, 123(3):227–233, 2007.

C. Green. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, page 9–18, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781450318235. doi: 10.1145/1281500.1281665. URL https://doi-org.tudelft.idm.oclc.org/10.1145/1281500.1281665.

E. Imhof. Positioning names on maps. *The American Cartographer*, 2(2):128–144, 1975. doi: 10.1559/152304075784313304.

G. W. Klau and P. Mutzel. Optimal labeling of point features in rectangular labeling models. *Mathematical Programming*, 94(2-3):435–458, 2003. doi: 10.1007/s10107-002-0327-9.

I. Pinto and H. Freeman. *The feedback approach to cartographic areal text placement*, volume 1121, pages 341–350. 01 2006. ISBN 978-3-540-61577-4. doi: 10.1007/3-540-61577-6_35.

N. Schwartges. *Dynamic Label Placement in Practice*. doctoralthesis, Universität Würzburg, 2015.

T. Strijk and M. Kreveld. Practical extensions of point labeling in the slider model. *GeoInformatica*, 6, 03 2000. doi: 10.1023/A:1015202410664.

M. van Kreveld, T. Strijk, and A. Wolff. Point labeling with sliding labels. *Computational Geometry*, 13(1):21–47, 1999. ISSN 0925-7721. doi: https://doi.org/10.1016/S0925-7721(99)00005-X. URL https://www.sciencedirect.com/science/article/pii/S092577219900005X.

P. van Oosterom and B. Meijers. Towards a true vario-scale structure supporting smooth-zoom. In D. Burghardt and M. Sesters, editors, *Proceedings of the 14th Workshop of the ICA Commission on Generalisation and Multiple Representation the ISPRS Commission II/2 Working Group on Multiscale Representation of Spatial Data: Geographic Information on Demand*, pages 1–19, 2011.

P. van Oosterom and B. Meijers. *Vario-scale data structures supporting smooth zoom and progressive transfer of 2D and 3D data*, pages 21–42. Nederlandse Commissie voor Geodesie - KNAW, 2012a. ISBN 978-90-6132-339-6.

P. van Oosterom and B. Meijers. True vario-scale maps that support smooth-zoom. In K. de Zeeuw, editor, *S.n.*, pages 1–9. Geospatial World Forum, 2012b. Geospatial World Forum - Theme: Geospatial Industryt and World Economy (Amsterdam, The Netherlands) ; Conference date: 23-04-2012 Through 27-04-2012.

J. W. van Roessel. An algorithm for locating candidate labeling boxes within a polygon. 16 (3):201–209, 1989. doi: 10.1559/152304089783814034.

A. Wolff, L. Knipping, M. van Kreveld, T. Strijk, and P. Agarwal. *A simple and efficient algorithm for high-quality line labeling*, volume 2001-44. Utrecht University: Information and Computing Sciences, uu-cs edition, 2001. ISBN 0924-3275.

A. Data-Preprocessing Workflow for tGAP Generation

This appendix documents the complete preprocessing pipeline that converts the original TOP10NL source layers into a topological data structure that can be used by the tGAP generator. Although not directly related to the main research topic of labelling vario-scale maps, the procedure is essential for replicability.

A.1. Merge thematic layers

After loading TOP10NL data into OpenJump, combine terrain, building, road and water features into a single layer. A data query can be used:

The resulting layer - hereafter called `dataset_name_merged` - contains all four feature classes.

Listing A.1: Combined Top10NL Dataset Query

```
SELECT naam,
       geom,
       visualisatiecode,
       'terrein'  AS original_layer
FROM   top10nl_terrein__terrein
UNION ALL
SELECT naam,
       geom,
       visualisatiecode,
       'gebouw'  AS original_layer
FROM   top10nl_gebouw__gebouw
UNION ALL
SELECT naam,
       geom,
       visualisatiecode,
       'wegdeel' AS original_layer
FROM   top10nl_wegdeel__wegdeel
UNION ALL
SELECT naamNL AS naam,
       geom,
       visualisatiecode,
       'waterdeel' AS original_layer
FROM   top10nl_waterdeel__waterdeel;
```

A.2. Create a planar partition

1. Open `dataset_name_merged` in OpenJUMP
2. Navigate to `Tools -> Edit Geometry -> Convert -> Planar Partition`
3. The operation outputs three layers:

   `Faces` — attributes: `FID`, `ID`

   `Edges` — attributes: `FID`, `ID`, `StartNode`, `EndNode`, `RightFace`, `LeftFace`

   `Nodes` — attributes: `FID`, `ID`

These layers encode the topology (nodes-edges-faces) required for tGAP.

A.3. Export layers to JML

Save each of the three layers plus the merged layer as JML files in the project working directory:

1. `dataset_name_merged.jml`
2. dataset_name_edge.jml
3. `dataset_name_face.jml`
4. `dataset_name_node.jml`

A.4. Convert JML to PostGIS dumps

Using GDAL/OGR, convert every JML file to a PostGIS-compatible SQL dump while preserving the Dutch national CRS (EPSG:28992):

Listing A.2: OGR2OGR Conversion Commands

```
ogr2ogr -s_srs EPSG:28992 -t_srs EPSG:28992 -f "PGDUMP" tmp_merged.sql
    dataset_name_merged.jml -lco GEOMETRY_NAME=geom -lco SPATIAL_INDEX=GIST
     -nln dataset_name_merged

ogr2ogr -s_srs EPSG:28992 -t_srs EPSG:28992 -f "PGDUMP" tmp_edge.sql
    dataset_name_edge.jml -lco GEOMETRY_NAME=geom -lco SPATIAL_INDEX=GIST -
    nln dataset_name_edge

ogr2ogr -s_srs EPSG:28992 -t_srs EPSG:28992 -f "PGDUMP" tmp_face.sql
    dataset_name_face.jml -lco GEOMETRY_NAME=geom -lco SPATIAL_INDEX=GIST -
    nln dataset_name_face

ogr2ogr -s_srs EPSG:28992 -t_srs EPSG:28992 -f "PGDUMP" tmp_node.sql
    dataset_name_node.jml -lco GEOMETRY_NAME=geom -lco SPATIAL_INDEX=GIST -
    nln dataset_name_node
```

Import the four tmp_*ṡql files with `psql` or pgAdmin.

A.5. Process topology tables inside PostGIS

Listing A.3: Topology Processing SQL Script

```
-- Rename attributes
DROP VIEW IF EXISTS dataset_name_edge_vw;
CREATE VIEW dataset_name_edge_vw AS
SELECT id               AS edge_id,
       startnode        AS start_node_id,
       endnode          AS end_node_id,
       rightface        AS right_face_id,
       leftface         AS left_face_id,
       (ST_DUMP(geom)).geom::geometry(LineString, 28992) AS geom
FROM   dataset_name_edge;

-- Generate a point-in-polygon (PIP) for faces
DROP TABLE IF EXISTS dataset_name_face_tmp;
CREATE TABLE dataset_name_face_tmp AS
SELECT id AS face_id,
       ST_PointOnSurface(geom)::geometry(Point, 28992) AS geom
FROM   dataset_name_face;

CREATE INDEX dataset_name_face_tmp_geom_idx
    ON dataset_name_face_tmp
    USING gist (geom);

-- Process nodes
DROP VIEW IF EXISTS dataset_name_node_vw;
CREATE VIEW dataset_name_node_vw AS
SELECT id AS node_id,
       (ST_DUMP(geom)).geom::geometry(Point, 28992) AS geom
FROM   dataset_name_node;

-- Enrich faces with semantic attributes from the merged layer
DROP TABLE IF EXISTS dataset_name_face_enriched;
CREATE TABLE dataset_name_face_enriched AS
WITH ranked AS (
    SELECT f.face_id,
           m.visualisatiecode   AS klass,
           m.naam,
           f.geom               AS pip_geometry,
           ROW_NUMBER() OVER (
               PARTITION BY f.face_id
               ORDER BY m.visualisatiecode ASC
           ) AS rn
    FROM   dataset_name_face_tmp AS f
    LEFT JOIN dataset_name_merged AS m
           ON  m.geom && f.geom
           AND ST_Contains(m.geom, f.geom)
)
SELECT face_id,
       klass,
       naam,
       pip_geometry
```

```
51 FROM   ranked
52 WHERE  rn = 1;
```

Export the two views dataset_name_edge_vw and dataset_name_node_vw to PostGIS dumps so they can be re-imported as tables:

Listing A.4: OGR2OGR Export Commands

```
1  ogr2ogr --config PG_USE_COPY YES -f "PGDUMP" tmp_edge.sql \
2          PG:"dbname=tgap_test user=postgres ..." \
3          dataset_name_edge_vw \
4          -nln dataset_name_edge \
5          -lco CREATE_SCHEMA=OFF -lco SCHEMA=public -lco GEOMETRY_NAME=
               geometry
6
7  ogr2ogr --config PG_USE_COPY YES -f "PGDUMP" tmp_node.sql \
8          PG:"dbname=tgap_test user=postgres ..." \
9          dataset_name_node_vw \
10         -nln dataset_name_node \
11         -lco CREATE_SCHEMA=OFF -lco SCHEMA=public -lco GEOMETRY_NAME=
               geometry
```

A.6. Build minimal bounding rectangles (MBRs) for faces

Listing A.5: Minimum Bounding Rectangle (MBR) Generation

```
1  DROP TABLE IF EXISTS dataset_name_mbrs;
2  CREATE TABLE dataset_name_mbrs AS
3  SELECT face_id,
4         ST_SetSRID(ST_Envelope(ST_Collect(mbr_geometry))::geometry(Polygon),
               28992)
5           ::geometry(Polygon,28992) AS mbr_geometry
6  FROM (
7        SELECT left_face_id  AS face_id,
8               ST_Envelope(ST_Collect(geometry::box2d)) AS mbr_geometry
9        FROM   dataset_name_edge
10       WHERE  left_face_id  IS NOT NULL
11         AND  left_face_id  > 0
12       GROUP  BY left_face_id
13
14       UNION ALL
15
16       SELECT right_face_id AS face_id,
17              ST_Envelope(ST_Collect(geometry::box2d)) AS mbr_geometry
18       FROM   dataset_name_edge
19       WHERE  right_face_id IS NOT NULL
20         AND  right_face_id > 0
21       GROUP  BY right_face_id
22 ) AS mbr
23 GROUP BY face_id;
```

A.7. Assemble the final dataset_name_face table

Listing A.6: Final Face Table Creation

```
DROP TABLE IF EXISTS dataset_name_face;
CREATE TABLE dataset_name_face AS
SELECT m.face_id,
       f.pip_geometry        ::geometry(Point,   28992),
       m.mbr_geometry        ::geometry(Polygon, 28992),
       f.klass               AS feature_class,
       f.naam                AS name
FROM   dataset_name_mbrs      AS m
JOIN   dataset_name_face_enriched AS f
  ON   m.face_id = f.face_id;
```

The resulting tables - dataset_name_node, dataset_name_edge, and dataset_name_face - conform to the schema expected by tGAP.

This workflow can be repeated for alternative study areas or updates to the TOP10NL dataset by replacing file names and schemas with the desired *dataset_name*.

B. tGAP Schema Description

The following describes the schema of the two main tables used from the tGAP dataset: `dataset_tgap_face` and `dataset_tgap_edge`. These tables store topological data, where each face or edge has an associated validity range defined by `step_low` and `step_high`.

Only the key columns used in this thesis are described below:

Table: {dataset}_tgap_face

Column Name	Data Type	Description
face_id	integer	Unique identifier for each face (polygonal region).
step_low	integer	Lower bound of the step during which the face is valid.
step_high	integer	Upper bound of the step during which the face is valid.
area	numeric	Area of the face geometry.
feature_class	integer	Category of the face feature(visualization code).
mbr_geometry	geometry	Minimum bounding rectangle of the face geometry.
pip_geometry	geometry	A point that is inside the polygon.

Table B.1.: Schema of {dataset}_tgap_face table

Table: {dataset}_tgap_edge

Column Name	Data Type	Description
edge_id	integer	Unique identifier for each edge (line segment).
step_low	integer	Lower bound of the step during which the edge is valid.
step_high	integer	Upper bound of the step during which the edge is valid.
left_face_id_low	integer	face_id of the face on the left side of the edge during its valid period.
right_face_id_low	integer	face_id of the face on the right side of the edge during its valid period.
left_face_id_high	integer	face_id on the left after a topology change.
right_face_id_high	integer	face_id on the right after a topology change.
geometry	geometry	Geometry of the edge as a line string.

Table B.2.: Schema of {dataset}_tgap_edge table

This schema supports topology by tracking which faces are adjacent to each edge over steps, and conversely, which edges form a face during its valid step range. The validity of each geometric feature is managed using step ranges (`step_low`, `step_high`).

C. Reproducibility self-assessment

C.1. Marks for each of the criteria

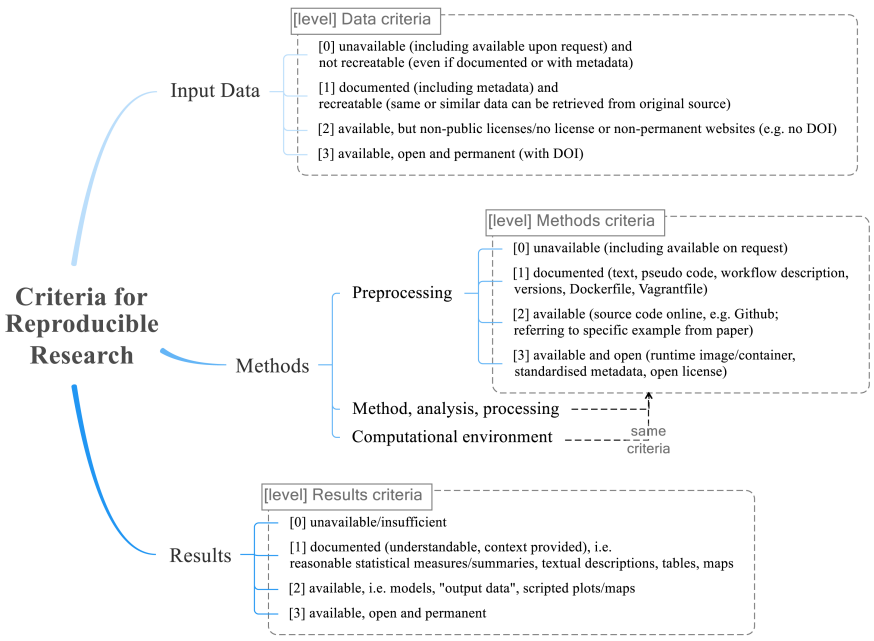


Figure C.1.: Reproducibility criteria to be assessed.

Table C.1 summarizes the reproducibility assessment of this thesis, based on the criteria outlined in Figure C.1.

Category	**Criteria**			**Mark**
	Available	**Open**	**Permanent**	
Input data	✓	✓	✓	3
Preprocessing	✓	✓	✓	3
Method, analysis, processing	✓	✓	✓	3
Computational environment	✓	✓	✓	3
Results	✓	✓	✓	3

Table C.1.: Reproducibility evaluation

C.2. Self-reflection

The reproducibility of this thesis was carefully considered. For the input data, all source data is obtained from the publicly available TOP10NL dataset. The pre-processing steps applied to this data are well documented in the Appendix A, ensuring transparency and the ability for others to replicate the preparation process. Regarding the methodology, the complete source code is openly available on GitHub, and an online demo has been provided to allow direct access and interaction with the implementation. Python 3.11 was used throughout the project, and all required dependencies are specified, making it straightforward to reproduce the results in a similar setup. A Node.js environment was used to set up the web interface of the application, supporting the development of the interactive demo.

Colophon

This document was typeset using LaTeX, using the KOMA-Script class `scrbook`. The main font is Palatino.