VISUAL ANALYTICS OF MULTIDIMENSIONAL TIME-DEPENDENT TRAILS

WITH APPLICATIONS IN SHAPE TRACKING

MATTHEW ANTHONY THOMAS VAN DER ZWAN

Cover: Visualization of plane trails over Europe, bundled using CUBu.

Visual Analytics of Multidimensional Time-dependent Trails
with Applications in Shape Tracking
Matthew Anthony Thomas van der Zwan
Thesis Rijksuniversiteit Groningen

ISBN 978-94-034-0851-4 (printed version)
ISBN 978-94-034-0850-7 (electronic version)

university of
groningen

Visual Analytics of Multidimensional Time-dependent Trails

with Applications in Shape Tracking

PhD thesis

to obtain the degree of PhD at the
University of Groningen
on the authority of the
Rector Magnificus Prof. E. Sterken
and in accordance with
the decision by the College of Deans.

This thesis will be defended in public on

Friday 5 October 2018 at 11.00 hours

by

Matthew Anthony Thomas van der Zwan

born on 11 September 1987
in Groningen

Supervisor
Prof. dr. A. C. Telea

Co-supervisor
Dr. M. H. F. Wilkinson

Assessment committee
Prof. dr. Lars Linsen
Prof. dr. Christophe Hurter
Prof. dr. Michael Biehl

ABSTRACT

Numerous problems in data science can be described under the common denominator of analyzing a set of trajectories, or trails, of objects moving in an embedding space. Two key classes of problems exist in this respect: First, we must devise methods and techniques to capture, or *track*, the motion of such shapes, starting from real-world sensor data such as video images. Secondly, we must devise methods and techniques to *analyze* and explore large sets of trails.

This thesis builds at the crossroads of the two above-mentioned problems. Our unifying concept is the representation of a trail (of a physical shape moving in Euclidean 3D space) as a multidimensional measurement, or data point. We first propose methods for tracking such shapes in the context of a concrete application – the localization of teats of cows from low-resolution 3D time-of-flight videos, with direct applications in the construction of automatic milking devices for the dairy industry. Secondly, we propose novel algorithms for the pre-segmentation of such video images, with the aim of localizing salient protruding shapes, such as teats, using shape skeletons – a novel way of representing and understanding grayscale and color images which generalizes the well-known concept of shape skeletons to continuous signals. Thirdly, we show how recent techniques for the visualization of multidimensional data can help understanding and improving the performance of complex computer-vision tracking algorithms for object trails, which is a novel way of utilizing visual analytics techniques. Finally, we show how truly large-scale trail-sets consisting of millions of static or dynamic trajectories of aircraft and eye-tracking data can be depicted in real-time and in a simplified manner, thereby addressing the question of large-scale analysis of trail data.

SAMENVATTING

Vele problemen in *data science* kunnen beschreven worden onder de noemer van het analyseren van een verzameling van trajecten van objecten die bewegen in de ruimte. Twee belangrijke types van problemen bestaan in deze richting: Men moet eerst methodes en technieken ontwerpen voor het *tracken* van de beweging van dergelijke objecten, op basis van actuele sensordata zoals videobeelden. Ten tweede moet men methodes en technieken ontwerpen voor het *analyseren* van grote verzamelingen van trajecten (*tracks*).

Dit proefschrift bestudeert de intersectie van de twee hierboven genoemde problemen. Het concept dat wij voorstellen ter vereninging ervan is de representatie van een traject (*trail*) van een fysieke eenheid die beweegt in 3D Euclidische ruimte als een multidimensionele meting of datapunt. We presenteren eerst methodes voor het tracken van dergelijke objecten in de context van een concrete toepassing – de localisatie van koeienspenen uit laagresolutie 3D *time of flight* video's, met directe toepassingen voor automatische melkrobots in de melkindustrie. Vervolgens presenteren wij nieuwe algoritmes voor de presegmentatie van dergelijke videobeelden, met het doel van localiseren van saillante vormen zoals spenen, met behulp van vormskeletten – een nieuwe manier van representatie en analyse van monochrome en kleurbeelden die het welbekende concept van vormskeletten generaliseert voor continue signalen. We laten zien, ten derde, hoe recente technieken voor de visualisatie van multidimensionele gegevens hulp kunnen bieden tot het begrip en verbetering van de effectiviteit van complexe computervisie *tracking* algoritmes voor objectpaden, een nieuwe toepassing van *visual analytics* technieken. Ten slotte laten we zien hoe daadwerkelijk zeer grote verzamelingen van paden van miljoenen statische of dynamische trajecten van vliegtuigen of oogbewegingen afgebeeld kunnen worden op een versimpelde en interactieve manier, wat het probleem van analyse van grootschalige padenverzamelingen addresseert.

PUBLICATIONS

This thesis is based on work that has appeared previously in the following publications (in order of relevance):

1. M. van der Zwan, V. Codreanu, and A. Telea. CUBu: Universal real-time bundling for large graphs. *IEEE TVCG*, 22(12):2250–2263, 2016
2. M. van der Zwan, Y. Meiburg, and A. Telea. A dense medial descriptor for image analysis. In J. Braz, S. Battiato, and F. Imai, editors, *Proc.* 8th *IEEE International Conference on Computer Vision Theory and Applications (VISIGRAPP)*, volume 1, pages 133–140, 2013
3. T. Klein, M. van der Zwan, and A. Telea. Dynamic multiscale visualization of flight data. In S. Battiato and J. Braz, editors, *Proc. of the* 9th *IEEE International Conference on Computer Vision Theory and Applications (VISAPP)*, volume 1, pages 232–240, 2014
4. M. van der Zwan and A. Telea. Robust and fast teat detection and tracking in low-resolution videos for automatic milking devices. In J. Braz, S. Battiato, and F. Imai, editors, *Proceedings of the* 10th *IEEE International Conference on Computer Vision Theory and Applications (VISAPP)*, volume 3, pages 654–667, 2015
5. M. van der Zwan, A. Telea, and T. Isenberg. Continuous navigation of nested abstraction levels. In M. Meyer and T. Weinkauf, editors, *Proc. EG/IEEE VGTC Conference on Visualization (EuroVis) – Short papers*, pages 13–17, 2012
6. M. van der Zwan, A. Telea, and T. Isenberg. Smooth navigation between nested spatial representations. In *Proceedings of the National ICT.OPEN/SIREN 2012 Workshop, October 22–23, 2012, Rotterdam, The Netherlands*, pages 140–144, 2012
7. M. van der Zwan, W. Lueks, H. Bekker, and T. Isenberg. Illustrative Molecular Visualization with Continuous Abstraction. *Computer Graphics Forum*, 30(3):683–690, May 2011

CONTENTS

1 INTRODUCTION

We live in a world that is densely populated with *shapes*. Our familiar surrounding universe consists of three-dimensional shapes representing both natural and man-made objects. Their structure, form, topology, properties, behavior, and mutual interactions create a rich and complex set of information that we can study to better understand the phenomena surrounding us.

In the last decade, significant advances in sensing devices, data storage size and speed, computational speed, and sophistication of algorithms have made it possible to both acquire more (and more diverse) data and analyze such data to extract information and, ultimately, knowledge. The above is also true for data and applications related to shapes. For example, acquiring live high-resolution video streams that capture shapes around us has become part of commodity realm. More recent developments, such as time-of-flight cameras, three-dimensional scanners, and tracker devices have brought the same simplicity, low cost, and high fidelity to the acquisition of three-dimensional static and dynamic content. Separately, increasingly sophisticated algorithms have been designed to analyze both static and dynamic information describing both 2D and 3D shapes [52]. Many such algorithms can now run at near-interactive framerates on consumer-grade personal computers or even on low form-factor mobile devices such as tablets and smartphones [296].

Collecting and analyzing information on shapes (and their behavior) should, of course, serve specific aims related to specific applications. These include, for instance, capturing the geometry and topology of real-world 3D shapes for synthetic reproduction via 3D printing [157]; searching for such shapes in large collections of pre-recorded exemplars, a process also known as content-based shape retrieval [261]; capturing the dynamic behavior of shapes, a process also known as shape tracking [306]; and analyzing the captured information to detect and extract higher-level knowledge of the structure and behavior of the surrounding world.

1.1 THE DYNAMICS OF SHAPE

A particularly interesting subfield of the above domain relates to shape *dynamics*. In this field, one is interested in patterns of *change* related to various measurable properties of shapes. Well-known applications in this field include tracking the motion of humans from video streams to identify anomalous behavior [197]; analyzing repetitive motion patterns to help athletes or patients in training and/or injury recovery [181]; record motion patterns of humans that can be next used to realistically animate synthetic characters in video games or special-effects movies [39]; detect congestion patterns in vehicle motion over roads, sea, or air so as to optimize traffic [304]; analyze motion of large groups of vehicles to detect potentially hazardous situations [122]; and let robots perform activities that involve interacting with real-world moving objects [55].

To accomplish the above goals, three classes of techniques are typically considered. First, *acquisition* techniques are used to collect actual data describing the moving shapes of interest. Secondly, automatic *analysis* techniques are used to extract higher-level information from the raw acquired motion data. Thirdly, *visualization* techniques are used to further refine the data analysis in a user-controlled way, and also to present the extracted information in ways that enables one to gather knowledge or insight on the underlying phenomena.

However, many challenges still remain open in the above endeavor. While data acquisition on the dynamics (motion) of 3D shapes is now a quite mature field, featuring ready-to-use techniques and tools, the analysis and visualization components of the process are, we argue, relatively less developed, as explained next.

Dynamic shape analysis: For the analysis part, many algorithms exist, which can be roughly classified into shape *detection* methods (which aim to separate a shape of interest from its surroundings in the acquired data), and shape *tracking* methods (which aim to extract the change of relevant shape properties over time). Both 3D shape detection and tracking algorithms are typically studied within the field of computer vision, the main reason for this being that the acquisition of high-resolution video streams depicting changing shapes is extremely simple, unexpensive, and non intrusive. Many computer vision techniques exist to these ends, ranging from the segmentation of simple rigid 2D shape silhouettes from static background in high-quality, high-contrast videos to the tracking of highly deformable, variable, and possibly occluded 3D shapes in low-quality videos [306].

Generally speaking, the complexity of a shape detection-and-tracking vision method is increasing as a function of the decreasing quality of the acquired video data; complexity and variability of shape and its changes; desired level of robustness, accuracy, and automation; and decreasing level of available computational power. When all above factors take extreme values, such as low-resolution or low-accuracy videos; handling organic, highly-deformable, and (partially) occluded shapes; the need for fully automatic, real-time, and computationally efficient algorithms, which are required *e.g.* for controlling robots by embedded devices; and guaranteeing tight tracking error bounds that a robot requires to properly operate, many of the existing state-of-the-art vision algorithms cannot be applied. To address such use-cases, new *custom* algorithms have to be designed.

Separate problems appear in the context of designing such new tracking algorithms, such as validation and optimization. To perform validation, one typically needs ground-truth information on the 3D tracked data. However, such information may not be available, or be very expensive to reliably collect. Separately, to perform optimization, one needs to intimately understand how the proposed tracking algorithm works, which are the parameter value-ranges for which it has issues, which are these issues, and how these can be corrected. In turn, this requires fast, simple, and above all intuitive ways that let designers explore the potentially very large, or high dimensional, 'state space' of a tracking algorithm.

Dynamic data visualization: The depiction of time-dependent data falls within the field of data visualization, which knows a long history, and has proposed many techniques [101, 271]. However, our context of understanding dynamic behavior of 3D shapes raises several specific challenges. First, the dynamic data at hand can be *abstract*, such as in the case of changes of internal parameters of a computer vision tracking algorithm. Secondly, the data can be *multidimensional*. For instance, understanding the behavior of a tracking algorithm requires understanding the dynamic changes of its input (*e.g.* a video stream), output (*e.g.* the degrees of freedom of the tracked 3D object(s)), internal tracker parameters which control its operation, and above all how all these variables *depend* on each other. Separately, understanding the dynamics of a *large and complex* set of objects, such as tens of thousands of vehicles moving over large spatial extents and time periods with variable direction, speed, height, and other properties, requires ways to show large amounts of information in a simplified manner.

1.2 FOCUS OF THIS THESIS

Summarizing the above, we can say that the analysis and understanding of large and complex time-dependent behavior of 3D shapes is a topic at the crossroads of computer vision and data visualization. We need computer vision to extract high(er) level information from raw acquired motion data; and we need visualization to show and interpret such higher level information. Furthermore, we can use visualization to understand the behavior of a tracker itself (rather than the tracked objects), so as to understand its limitations and next improve its operation. This last use-case falls within the scope of *visual analytics*, a relatively new discipline which focuses on the use of interactive visualization techniques to make sense of complex processes based on large amounts of complex data [276, 297].

Computer vision, information visualization, and visual analytics are all established disciplines – in this order. However, mainly due to historical reasons, they have evolved relatively separately, so using methods from one field (*e.g.* visual analytics and/or information visualization) to support another field (*e.g.* computer vision) is not yet mainstream. Recent example applications show, however, that important benefits can be gained by combining the three fields [13, 164, 212].

1.2.1 *Research Questions*

Given the above, we can now state the main research question of this thesis:

RQ: *How can visual analytics help understanding time-dependent multidimensional data to support the analysis of the dynamic behavior of 3D shapes?*

We will further split this question into two sub-questions, based on the relationship between *data* and *images*, as follows (see also Figure 1.1):

1. From images to data: Computer vision (tracking) algorithms can be thought of as algorithms reading images (video streams) and generating data (characteristics of the tracked objects). We are interested in the design of computer vision algorithms for tracking complex 3D shapes, under the constraints mentioned earlier in this section (complex/variable shapes, low-resolution or low-accuracy acquired data, full automation, high tracking accuracy, and low computational power). As mentioned, developing tracking algorithms under all such constraints is hard and expensive. As such, we ask ourselves

RQ1: *How can we use visual analytics to understand and improve the operation of fully automated, low-cost, computer vision tracking algorithms for complex 3D shapes from low-quality video data?*

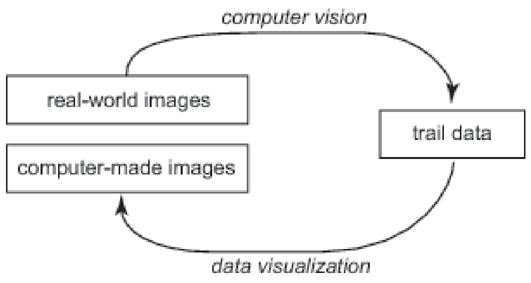


Figure 1.1: From images to data and conversely.

2. From data to images: Visualization algorithms can be thought of as algorithms reading data and generating synthetic pictures that help humans understanding the data. Our application context involves, as explained, data that is multidimensional, potentially abstract (non-spatial), time-dependent, and potentially large. Understanding such data can help users to understand the underlying phenomena, be them either the behavior of 3D dynamic shapes or the behavior of a tracker for such shapes. As such, we ask ourselves

RQ2: *How can we develop information visualization techniques that help us effectively getting insight in multidimensional, spatial or non-spatial, time-dependent, and potentially large trail datasets?*

Given the above, the remainder of this thesis is naturally split into two parts, each of them covering the research questions *RQ*1 and *RQ*2. These parts, including a specific application demonstrating the relevance of the respective research questions, are detailed further below in Sections 1.3 and 1.4 respectively.

Before going into the elaboration of *RQ*1 and *RQ*2, let us however clarify a central data concept that links them: the trail.

Trails: The data involved in all our use-cases discussed above has several special characteristics: it is time-dependent; it has multiple values per measurement (it is multidimensional); and it is large, complex, or both. Besides these structural properties, our data has a much more important semantic common denominator: It denotes trajectories, or paths, taken by objects as these move through their embedding space. We next refer to such trajectories as *trails*.

Summarizing the above, the following important properties of trails are relevant to our work:

- **Time-dependence:** All measurements along a trail relate to properties of the tracked object (that is, object that moves along a trail) that are sampled, or measured, at consecutive moments in time;
- **Multidimensionality:** At any given time moment, the tracked object has several properties. Each of these spawns a different dimension;
- **Embedding space:** Properties measured along a trail belong to different spaces. For example, if we consider the motion of a shape in three dimensions, its measured properties are related to how the object moves in 3D space. However, if the object is some abstract data entity of which none of the measured properties are spatial, then its trail (as it changes in time) are not directly related to physical (2D or 3D) space. Note that the embedding space is a chosen, not a given, aspect of change: We can, for instance, examine the properties of a physical shape, tracked by a computer-vision tracking algorithm, as this shape moves in its natural embedding physical space (2D or 3D). Alternatively, we can examine the shape's properties from the perspective of the tracker algorithm's parameter changes, which do not (necessarily) constitute a physical (2D or 3D) space;
- **Volume:** Depending on the application, we can have few trails, such as in the case we are tracking one or a few shapes as they move through 3D space. Examples are tracking specific features of a single natural object, examples of which are given next in Sec. 1.3. However,

we can also track a large set of objects, or compare many tracks of the same simple object under various configurations; examples are given in Secs. 1.3 and 1.4. This creates a large variability in the number of tracked *objects*, which in turn creates various visualization and analysis challenges. Separarely, we can, during tracking of a single shape, consider few, or many, properties of that shape as it changes in time. This creates a large variability in the number of tracked *dimensions*, or variables. Examples are, again, given in Secs. 1.3 and 1.4.

All above aspects contribute to subsequent challenges that relate to both *RQ*1 and *RQ*2. These challenges we will describe next.

1.3 VISUAL ANALYTICS FOR TRACKER DESIGN

Computer vision tracking systems exist in many flavors and for many types of problems (shapes, input images, type of tracked properties, and usage context). As such, it is not realistic to approach solving *RQ*1 in a general sense. To provide a higher added-value to the approach and solutions we will propose next, we need a concrete use-case which meets the various constraints and challenges listed earlier in the formulation of *RQ*1. Such a use-case, embodied in a real-world industrial context which also kickstarted our research, is outlined next.

1.3.1 *Use-case: Automatic milking devices*

Over the last decades, all industries have seen a gradual move from manual labor to mechanical labor with increasing levels of automation. One industry branch where this process is relatively more recent, and as such not yet finalized and still open to research, is the dairy industry. Within this industry, an important cost-component is the milking of cows. Historically, this happened by hand. Several decades ago, automatic pumping devices emerged, which accelerated a part of the process. However, most such devices still require a human operator (the farmer) to manually attach the suction cups to the cow.

In the process of further automating the milking, so-called *automatic milking devices* (AMDs) have emerged. These devices attempt to attach themselves to the cow's teats in an automatic way. If this can occur fully automatic, the manual labor component is significantly decreased, leading to lower costs and/or higher yields [138].

A key component of an AMD is the detection of the teats of a cow's udder, so that its moving part, typically a robotic arm, can attach the milk suction cups to them. A second important component involves tracking these teats, as a cow cannot be immobilized during the milking process. In early AMD's, both detection and tracking have relied on sensing devices such as laser scanners and standard optical cameras. However, such devices cannot operate reliably in a typically dusty and dark stable environemnt, and they are also mechanically sensitive. Recently, stereo or three-dimensional (time-of-flight) cameras have become available at framerates, form-factors, and prices which make them an attractive alternative to explore. However, such devices come with their own challenges – low resolution, sparse scene acquisition in terms of a point cloud, and limited depth accuracy [138].

For our use-case, we thus aim to design an efficient and effective front-end algorithm for teat detection and tracking based on a 3D time-of-flight camera. To be effective, such a solution needs to work fully automatic (in order to control the autonomous milking robot), have high accuracy (so as to steer the milking cups to precisely latch on the teats), be fast (so as to handle spurious and abrupt movements of the cow), and use limited computational power (so it can be implemented on low form-factor hardware that can be mounted on the AMD). Clearly, all above requirements match well the context of *RQ*1. The selection of this type of use-case is motivated by the interest in this type of application of Lely Technologies [147], a forefront company in the Netherlands in the area of AMD construction, which was a key supporter of the research described in this thesis.

At first glance, the tracking problem we face is relatively simple, given the state-of-the-art of research in computer vision. However, most such research is driven by conditions that do not apply to our context, *e.g.* the availability of high-resolution and/or good-contrast color images; the availability of high computational power; the intervention of users for initialization or calibration of parameters; and a constrained relative position of the camera and the tracked shape.

To address our problem, we first study existing 3D tracking techniques, and determine which ones are potentially applicable to our context (Chapter 2). Next, we propose a novel tracking algorithm that complies with our context's constraints (Chapter 5).

1.3.2 *Verification, validation, and improvement of AMD tracker*

Every industrial system needs to be verified and validated before it gets deployed. This also holds for our tracking system, given its intended use in an industrial context, and even more specifically so in a robotic device. Additionally, the intended use of our teat tracking system will have to deal with live animals in an unsupervised context. As such, the system should robustly detect the teats of the cow to be milked, so as to steer the AMD robot precisely towards the target. Failure to do so may result in injury to the animal and/or damage to the robot.

Tracking systems for (3D) real-world shapes are often verified and validated using ground-truth data which describes where in the image (sequence) an object is located. For our case however, this data is not readily available: We do not know where, in a 2D image acquired by our imaging device, the teats of a cow to be milked are precisely located. The only way to generate such ground-truth data is to *manually* annotate video sequences acquired in the field to indicate the teat locations, if teats are visible. However, generating such manual annotations for tens or hundreds of videos, each containing hundreds up to thousands of frames, is clearly not practical.

Hence, we need to investigate other ways to gauge the accuracy of our proposed tracker. For this, we propose a set of visual analytics methods which examine and display the entire so-called 'state' of our proposed tracker, including its input information (video sequence), output information (3D positions of the tracked teats), and internal variable values (used during the tracking process). By examining variations of such variables, and correlations and outliers thereof, we show how the high-level behavior of the tracker can be understood by a system designer, so that problematic configurations can be spotted and eventually alleviated, and without necessarily availing of ground-truth data. The aim of our visual analytics system is to cover the entire spectrum ranging from examination of a single tracking sequence (video), detailed frame-level investigation of parameter values, up to the global overview-analysis of a large set of different tracking sequences. This way, we aim to cover both low-level defect detection and removal and broad statistical evaluation of the tracker by means of a large sequence of test runs. Our proposed visual analytics system, together with the insights we derived from its use in terms of understanding and alleviating the limitations of our tracking system, are presented in Chapter 6.

1.4 VISUAL ANALYTICS FOR LARGE MULTIDIMENSIONAL DYNAMIC TRAIL DATA

The second research question of our thesis (*RQ*2) takes our focus from the 'micro' to the 'macro'. That is, *RQ*2 focuses on answering the question of how we can analyze (and depict) a large set of multidimensional trails – much larger than the ones delivered when considering *RQ*1 – in ways that allow us to understand similarities, outliers, and correlations of measured variables. Separately, we focus here on the problem of understanding the tracked information, rather than on the problem of improving the data-acquisition process (tracking method) which delivers us the information. As such, *RQ*2 is concerned much more with information visualization questions rather than computer vision problems.

Similarly to *RQ*1 (Sec. 1.3), we need to consider a concrete use-case for supporting our research. Given the focus of *RQ*2 on very large data volumes, we cannot (arguably) use information obtained from computer vision methods. In particular, we cannot use the AMD context, since this delivers only tens or possibly hundreds of moving shapes (cows) over relatively short periods of time (minutes). Hence, we turn to a quite different source of data: Trails of large collections of moving vehicles over large periods of time.

1.4.1 *Use-case: How airplanes move*

There are many sources for such data. One of them is the motion of airplanes over large spatial regions and long time periods [207]. This context can provide us with tens of thousands up to millions of moving objects over periods of time ranging from days to months.

Given the sheer size of the data, and also its availability from third-party sources (we cannot track planes ourselves), the issue of analyzing tracking-algorithm parameters for optimization purposes becomes now far less relevant. The challenge, in contrast, is how to display such sheer amounts of data so that interesting space-time phenomena become easily visible.

To attack this challenge, we proceed as follows. First, we review information visualization and visual analytics methods for large dynamic multidimensional datasets (Chapter 3). Next, we present a novel method that addresses this type of analysis, with usage examples for our considered use-case – the analysis of large sets of airplanes moving over time (Chapter 8).

1.5 STRUCTURE OF THIS THESIS

Summarizing the above, the structure of this thesis is as follows.

Chapter 1, the current chapter, presents the scope of our work, which lies at the crossroads of *acquiring* and *analyzing* motion trails of 3D shapes, using visual analytics methods.

Chapter 2 presents related work on the design of computer-vision trackers for 3D shapes. This outlines state-of-the-art and how it does (or does not) match the requirements of our AMD cow-milking application outlined in Sec. 1.3.1. This chapter relates to *RQ*1.

Chapter 3 presents related work on the visual analysis of multidimensional time-dependent trail data. This outlines state-of-the-art in this field and how it does (or does not) match the requirements of our large-scale airplane motion analysis application outlined in Sec. 1.4.1. This chapter relates to *RQ*2.

Chapter 4 presents our work towards the robust segmentation of shapes from their surrounding backgrounds, which is strongly related to the problem of tracking shape motion, as outlined earlier in Sec. 1.1. We present here a novel method that addresses this segmentation problem, and discuss its suitability for our specific shape-tracking problem (see Sec. 1.3.1) and also in the wider context of segmenting arbitrary shapes from 2D images.

Chapter 5 presents our solution to the AMD cow-teat-tracking problem outlined in Sec. 1.3.1. We introduce here several novel algorithms whose main feature is complying with the complex set of requirements posed by their application on AMD robots (Sec. 1.3.1). This chapter relates to *RQ*1.

Chapter 6 presents our visual analytics solution for the examination and interpretation of the data produced for the AMD tracker proposed in Chapter 5. We introduce here several visual analytics techniques for the overview investigation of tracker data, detection of outlier events (tracking challenges), comparison of tracking sequences, and explanation of the tracking problems. This chapter relates to *RQ*1.

Chapter 7 presents a novel information-visualization solution for the display and visual analysis of very large collections of trails. The presented technique is between one and two orders of magnitude computationally faster than all state-of-the-art techniques we are aware of. It is also application-agnostic, meaning, it can be used for the simplified visualization of massive spatial trail datasets coming from any application domain. This chapter relates to *RQ*2.

Chapter 8 presents an adaption of the techniques introduced in Chapter 7, and validation thereof, for a concrete problem – the examination of a large set of airplane trails for air-traffic-control (ATC) purposes. This chapter plays for *RQ*2 a similar role as Chapter 6 played for *RQ*1.

Chapter 9 concludes this thesis. Here we compare our techniques and proposals for addressing both *RQ*1 and *RQ*2, reflect on their limitations, and outline potential directions for future work concerning the (as we have seen) joint topic of shape tracker design and visualization of shape trails.

2 RELATED WORK: SHAPE TRACKING

The design of algorithms for tracking 3D shapes is one of the main research directions in the field of computer vision. Since this is also an integral part of our research question outlined in Chapter 1, we next provide an overview of related work in this area. In this overview, we also identify the challenges that arise when trying to apply computer vision techniques to our proposed application of automatic milking devices (AMD's). Since tracking algorithms usually consist of two steps, one for finding the shapes of interest in each image of a video and a second step for keeping track how such shapes move over time, we discuss such methods in separate sections. The structure of this chapter is as follows: Section 2.1 provides definitions and introduces properties of tracking methods that are relevant throughout our work. Section 2.2 discusses object detection algorithms. Section 2.3 discusses tracking algorithms.

2.1 PROPERTIES OF TRACKING METHODS

In their noteworthy survey paper on object tracking, Yilmaz et al. [306] define three key steps in video analysis: Detection of interesting (moving) objects, tracking of such objects from frame to frame, and analysis of tracks to recognize their behavior. At first glance, our first research question (RQ1) is mostly concerned with the last of these steps. However, we also aim to use the information gathered during the analysis of tracks to (detect ways to) improve the object detection and tracking, thereby completing the visual analytics cycle.

Figure 2.1 shows how the steps of object detection and object tracking are connected in the object tracking pipeline, also including a temporal component by referencing the tracking result of the previous frame. Here and next, we use the following notations: $\tilde{\Omega} \subset \mathbb{R}^3$ is an actual (physical) shape; I is a two-dimensional image of $\tilde{\Omega}$, captured by a sensing device, such as a video camera; $\Omega \subset \mathbb{R}^3$ is the shape that the tracking recovers from I; and, for all

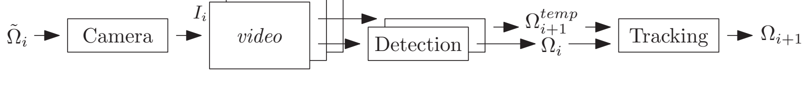


Figure 2.1: Generic object tracking pipeline. From left to right we see the actual object $\tilde{\Omega}_i$; its images I_i obtained by a camera; the detected shapes Ω^{temp}_{i+1} from these images; and the way tracking uses a detected shape Ω^{temp}_{i+1} from the current frame with a tracked shape Ω_i from the previous frame to yield the tracked shape Ω_{i+1} for the current frame.

above, a subscript identifies a frame, or time-moment, when the respective quantities have been measured or computed. Within this framework, detection is responsible for capturing the *spatial* embedding of the object $\tilde{\Omega}$ and tracking captures the *temporal* variation of $\tilde{\Omega}$. In the remainder of this chapter we discuss relevant related work accordingly. Next, in chapter 5, we present our tracking solution following these notations and framework.

During each step of tracking our objects, we need a way to represent, or model, the objects we deal with. The way we choose to represent our tracked objects also constrains the methods that can be used for tracking; indeed, some tracking methods require information that is not provided by certain shape representations. Separately, the targeted application may also have constraints on the object representation; for instance, if we want to track an object's orientation, we need to represent the object by more than its position.

Besides the way one models the shape of an object, *i.e.*, the surface $\Omega \subset \mathbb{R}^3$, the object's appearance can also be important. Generically, appearance can be described by a multivariate function $a : \mathbb{R}^3 \to \mathbb{R}^n$ that associates to any point $\mathbf{x} \in \partial\Omega$ of an object's surface a (n-dimensional) vector of properties, such as color, brightness, shading, texture, or surface normals. Appearance is essential for both object detection, since we can separate an object's silhouette from its surroundings only if the two areas have some appearance-related differences. Appearance is also essential for object tracking, since we can say that a certain part π of an object moved that much between frames i and $i+1$ of a video only if the image fragments $I_i(\pi)$ and $I_{i+1}(\pi)$ related to the part π are relatively similar. Finding such similar image fragments is a major challenge in computer vision known under the names of correspondence computation or image matching [305].

In the remainder of this section we first provide an overview of commonly employed shape representations, followed by a similar overview of appearance representations, both related to shape detection and tracking. Both overviews follow the survey in [306].

2.1.1 *Object representations*

Object representations aim at capturing the geometry (or shape) of an object, and the embedding thereof (most commonly known as orientation and position) in the surrounding 3D space. Several object representation techniques exist for this, as follows.

POINTS Arguably the simplest way to represent an object Ω is using a single point, or a collection of points $P = \{\mathbf{x}_i\} \subset \mathbb{R}^3$, also known as a point cloud. Single-point representations are common to model the centroid of an object, as shown in Figure 2.2(a) [284]. Point cloud representations are more powerful as they can model local shape as well as orientation and size, as shown in Figure 2.2(b) and [232].

PRIMITIVE GEOMETRIC SHAPES Objects can also be represented using simple primitive shapes such as rectangles or ellipses and their 3D equivalents [47]. Examples can be seen in Figure 2.2(c, d). These representations strike a good balance between simplicity and computational efficiency and ability of modeling a shape's so-called extrinsic parameters, such as position, orientation, and scaling in the embedding space. Although the nature of primitive geometric shapes makes them more naturally suited for representing rigid objects, they are also used for representing non-rigid objects for tracking.

ARTICULATED SHAPE MODELS These representations aim at capturing more complex *deformations* of a shape during its temporal evolution, which cannot be captured effectively by simple rigid primitives, nor efficiently by point clouds, respectively. Articulated shape models decompose an object in multiple smaller parts which are linked by joints. Parts can be next represented by aforementioned methods, *i.e.*, point clouds or primitive geometric shapes; joints encode the allowed deformations of the entire object. To do this, one usually defines some restrictions in the form of, for example, kinematic motion models. Figure 2.2(e) shows the application of articulated shape models using geometric primitives as representation for the different parts of a human body. More details on articulated shape models can be found in [248] and related papers.

OBJECT SILHOUETTES AND CONTOURS In some cases, we want to capture the full outline of the object instead of an approximation. In this case, we use the *boundary* of an object, also called its contour. This can be represented as the full contour (Figure 2.2(h)) or as control points on the contour (Figure 2.2(g)). When we also include the region inside the contour, we call this the silhouette of the object (Figure 2.2(i)). Both representations can be used for tracking complex nonrigid shapes [305].

SKELETAL MODELS Besides contour and silhouette models, another way to represent the shape of an object is using its skeleton (see Figure 2.2(f)) which can be extracted from the object silhouette using a medial axis transformation [16]. The skeleton representation is often used in object recognition [4] and object retrieval [48, 49, 84, 301], but it can be used to model articulated and rigid objects for tracking as well. Skeletal representations, augmented with distances to the corresponding silhouettes, are shown to be *dual* shape representations – that is, they encode as much information as a boundary representation does [259]. As we will show in chapter 4, skeletal descriptions can also be used to perform image segmentation, a key step in shape detection.

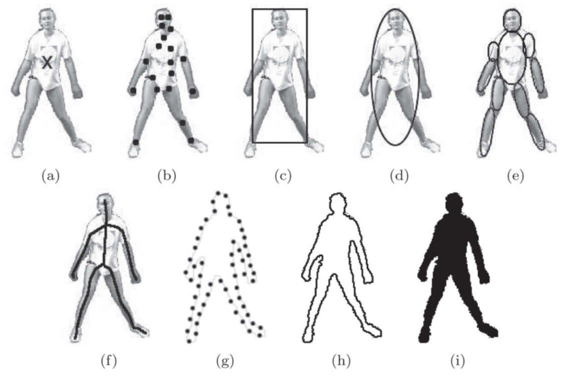


Figure 2.2: Different ways to represent an object, in this case a human body, from [306]: (a) shows the use of a single (centroid) point, whereas (b) shows how multiple points can be used to describe the same shape. Both (c) and (d) show the use of a simple geometric primitive as descriptor. In (e) we see an articulated shape model represented by geometric primitives. The contour of the object is represented using points in (g) and represented as continuous outline in (h). Finally, we see the object silhouette in (i) with the corresponding skeleton in (f).

2.1.2 *Appearance representation*

Now that we have seen how the object shape can be represented, we will take a look at the different ways to represent object appearance.

PROBABILITY DENSITIES The probability density of object appearance is based on object local features, such as color or texture, which can be computed from an image region defined by the object shape model (as long as this describes an image patch with sufficient area, as is the case with the geometric shape and contour/silhouette representation). To determine the probability density of an object, one can use parametric methods, *e.g.* Gaussian [311] or a mixture of Gaussians [196]. However, non-parametric probability density estimation methods such as Parzen windows [71] or histograms [47] can also be used.

TEMPLATES Templates are created from simple geometric shapes or object silhouettes [85] which simultaneously encode local shape and appearance of the desired objects. While templates are simple and fast to implement and use, they typically only represent the object appearance from a single viewpoint. Hence, single templates are not suitable for tracking objects whose appearance is expected to (drastically) change during tracking. However, as we shall see in Chapter 5, 2D and especially 3D templates can be very effective and efficient instruments for tracking constrained shape families such as present in our AMD context.

ACTIVE APPEARANCE MODELS Active appearance models are another way to jointly describe the object shape and its appearance [68]. The object shape is usually defined by a set of so-called landmarks which are points located in areas where one is specifically interested to describe the object for the underlying application. An appearance vector is used to store color, texture and brightness gradient magnitude for each landmark. Before being able to use an active appearance model, one needs to train the model to recognize the specified landmarks. As such, active appearance models share similarities with feature and keypoint extraction techniques such as SIFT [160] and SURF [19].

MULTIVIEW APPEARANCE MODELS To alleviate the problems of single-view appearance models indicated above, multiview appearance models describe an object from different views. This can be done by generating a subspace from the available views [25, 182], but also by training a set of classifiers to find locations of keypoints that match these multiple views [14, 197].

2.2 OBJECT DETECTION

Before a tracking method can start tracking objects, one needs to know where the objects to be tracked are located – in simple terms, we cannot track something unless we know what that something is (which means, in our context, where that something is). For some tracking methods, it is sufficient to detect the objects when they enter the image, after which tracking can proceed without repeating the detection. In cases when (the appearance of) the tracked object changes significantly between frames, one needs an object detection method to be run every frame. Independent of which kind of tracking is used, it is clear that a suitable object detection method is a requirement for successful tracking.

We next discuss several classes of object detection techniques in the context of shape tracking.

2.2.1 *Background Subtraction*

The simplest divide that can be made when describing an image is that between foreground and background. In this case, we define the foreground as the object of interest. While this is a simple concept, its actual implementation can be quite tricky. For example, say we want to detect a person walking along a street. In this case, we only want to detect the moving person, not the moving branch of a tree that is also in the image. Therefore, we cannot define background as those pixels $\mathbf{x} \in I$ which do not significantly change between consecutive frame I_i and I_{i+1}. More involved solutions have been proposed to deal with such situations, as discussed next in this section.

What most, if not all, background subtraction methods have in common is that they partition an image frame I into a set of foreground pixels $F \subset I$ and a disjount set of background pixels $B \subset I$, $F \cup B = \emptyset$, such that the shape of interest $\tilde{\Omega}$ is located (nearly) entirely in F. When the foreground set F is fragmented, connected component or morphological dilation filters can be used to 'bridge' small-scale holes and/or ignore small-scale components.

An early, but relevant, example of background subtraction is given by Wren et al. [298]. Here, background subtraction is applied to detect a single human body in front of a (relatively) static background. This is done by creating a Gaussian model for the color of each pixel in the image, using several consecutive frames to determine the model parameters. After the model computation is performed, a pixel is said to be part of the background if its likelihood is in correspondence with the model, otherwise the pixel is said to be part of the foreground. While this method is a good fit for its intended scenario – detecting a human body against a static background – the method is not suited for tracking against more dynamic backgrounds, such as our AMD udder-tracking scenario.

To be able to deal with dynamic backgrounds, Stauffer and Grimson [251] propose the use of a mixture of Gaussians to model pixel color. They also introduce an update scheme for the mixture of Gaussians to be able to deal with changes over time in a non-destructive way, i. e., when a background pixel changes to a different color for a while and then changes back. In such cases, the former Gaussian model is still present and is used again if there were no other changes in the meantime. The labeling of a pixel as foreground or background is based on matching with all available Gaussians. A pixel is said to match with a distribution if it is within a certain threshold of the distribution. If the matching Gaussian has enough support within the mixture, the pixel is said to be part of the background.

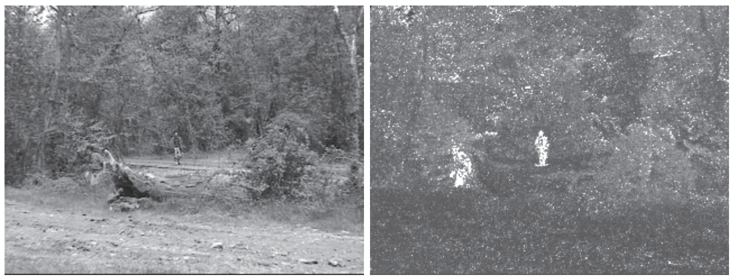


Figure 2.3: Example of background detection in action. Left: the input image; right: pixel values are assigned the inverse probability of belonging to the background (dark = background, bright = foreground). Taken from [72].

Instead of only relying on color information, it is also possible to incorporate more information in the background model so as to increase its accuracy. For instance, the background models of neighboring pixels can be used when labeling a pixel [72]. Similarly, texture features can also be incorporated into the model, reducing sensitivity to changes in lighting [156].

However, this changes the results to be region-based instead of pixel-based. Simply put, one uses spatial filtering to reduce noise-related effects, but the price to pay is that the foreground-background border becomes then globally as accurate as the filter radius.

The background subtraction methods listed above are all based on increasingly complex statistical models of the background. Another possible approach to the problem of background subtraction is to use Hidden Markov Models to describe the state of a pixel. For example, Rittscher et al. [219] apply Hidden Markov Models for background subtraction in the case of tracking cars on a highway. They also introduce a third state to indicate if an image segment is part of the shadow of an object. Stenger et al. [252] introduce a technique that can perform online training of a Hidden Markov Model with changing topology, i. e., the method is capable of changing the number of states that an image pixel is part of.

Background subtraction is a powerful method given the right circumstances. However, an important (implicit) requirement of background subtraction techniques is that the camera is stationary, or should only make small movements between frames. This means that background subtraction is probably not a suitable technique for detection in our intended application, as the camera is mounted on a moving arm which can move relatively fast. On the other hand, given that we are dealing with distance images, a simple threshold based background subtraction method could be a good pre-processing step before another object detection method is applied.

2.2.2 *Segmentation*

Where the background subtraction methods described above immediately assign meaning to the results of processing the image, i. e., foreground for the objects we are interested in and background for the rest, *segmentation* takes a different approach. The goal of a segmentation methods is to divide the image into smaller parts, called segments, where each segment consists of pixels which are similar to each other. While there are certainly a lot of ways to define pixel similarity, the important point here is that applying segmentation to an image usually results in more than the two (or three) classes generated by background subtraction. Furthermore, assigning meaning to the segments, i. e., determining which segments are part of the object we are interested in, is performed as an additional step after segmenting the image. Globally put, segmentation can be seen as a generalization of background subtraction; it is more powerful than background subtraction as similarity within a segment can be defined more flexibly, but it typically generates more segments than background subtraction, so it requires an additional segment interpretation step to identify which segments correspond to the moving shape $\tilde{\Omega}$ that we wish to track.

Image segmentation is an extremely rich field, with hundreds of proposed methods, and tens of application areas ranging between medicine, biology, document processing, surveillance, robotics, and image compres-

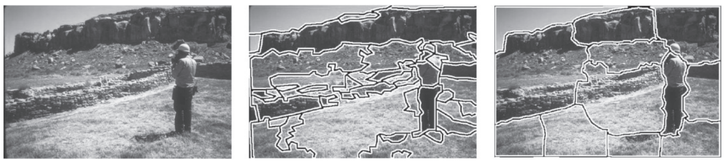


Figure 2.4: Example of image segmentations. From left to right: The original image, image segmented using mean-shift, and image segmented using normalized cuts. Images taken from [306].

sion [89, 90, 308, 309]. As the content (and even the titles) of the previously cited papers hint, the segmentation area is huge. It is impossible for us to give an overview thereof here. As such, we will next limit ourselves to outline the methods which have a direct connection with our own research described in the following chapters.

Mean shift: One of the best known algorithms for generic image segmentation is the mean-shift algorithm [46]. The mean-shift algorithm itself originates in the field of statistics and has many other applications, for instance in data clustering, or more specifically, in the context of simplified visualization of trail-sets, as we will demonstrate in chapter 7. The essence of mean shift for image segmentation is quite simple: Given an image I, each pixel $\mathbf{x} \in I$ is regarded as a point in a high-dimensional space, where the dimensions comprise the pixel's (RGB or similar) color and, optionally, the pixel's 2D coordinates. The choice of color space is important here, since we ultimately want to group pixels which look similar to humans. For this, Comaniciu and Meer [46] suggest using the L^*u^*v-space. Given this high-dimensional scatterplot of data points, mean shift next estimates the points' local density (using kernel density methods), and next shifts (moves) points in the direction of the density's gradient. This essentially 'condendes' the points around their local means. If one next finds the locations of these local means, one has readily identified all pixels that belong to a segment. Mean shift is quite effective and simple to implement. However, controlling the number of resulting segments, and applying the method in a computationally efficient way for even reasonably-sized images, is challenging.

Graph cuts: The second category of segmentation methods we discuss here is that of graph cuts [299]. Although different from the mean shift algorithm in many ways, it has in common that it also starts with a transformation to a different image representation. For graph cuts, the image is represented as a graph where pixels are vertices (nodes) and edges are inserted for all pairs of neighbor pixels with an associated edge weight based on the similarity of the pixels. To segment an image, a set of edges is removed from the graph so that two disconnected graphs remain. The set of removed edges and its associated cost are usually referred to as the *cut*.

By minimizing the cost of the cut, which is described in terms of the pixels' similarities, optimal segmentations can be achieved. The graph cut segmentation originally proposed by Wu and Leahy [299] uses a cost defined as a summation of the weights of the removed edges. As discussed there and in subsequent papers, this gives an undesired bias to smaller segments – in other words, oversegmentation often occurs. An improved metric called *normalized graph cuts* [237] has been proposed to fix this and other shortcomings of the original graph cut segmentation method. The cost function can also be modified to take into account the depth information created by a time-of-flight camera as shown by Arif et al. [11], thereby rendering this method interesting for segmenting video images as produced in robotics contexts similar to our AMD context. However, in general, controlling the size, shape, and number of segments generated by graph cuts methods is hard for low-resolution, high-noise, images.

Image Foresting Transform: Another image processing method based on graphs that can, amongst other applications, be adapted to perform image segmentation is the Image Foresting Transform (IFT) [82]. The image representation used by the IFT is similar to that used by graph cut methods: Image pixels become nodes and the edges represent similarity-encoding connectivity between pixels. The IFT can be used to design, implement, and evaluate connectivity-based image processing operators. This is achieved by defining the a minimum cost function corresponding to the operator and compute a minimum-cost path forest. IFT has been used to deliver effective and efficient segmentations of complex images, with notable applications in the medical domain [81]. Yet, to our experience IFT-based methods work best in contexts where the user can steer the actual segmentation – a scenario which is not applicable to our fully-automatic AMD context.

Other methods: the goal of all segmentation methods is to subdivide the image in (hopefully) meaningful segments. Often, we are interested in the contour, or boundary, of a segment as well as (or even more than) the segment itself. Instead of first computing the segments and then find the contours, it is also possible to do it the other way around. The *gPb-owt-ucm* method [10] segments an image by first determing the image contours using the *gPb* contour detector [166] and applying a modified watershed transform [220] on this image. One of the advantages of this approach is that the produced segmentation is hierachical, allowing the user to select the appropriate level using a single level-of-detail parameter. Still, such methods do not (entirely) cope with our fully-automatic constraints.

Given the right parameter tuning and choice of algorithm, the segmentation methods can be used to separate the object of interest from the rest of an image. However, after performing segmentation, we still need to identify which of the segments corresponds to our object, for example, by comparing characteristics of the object of interest and the segment(s). Therefore, like background subtraction, segmentation can also be viewed as a pre-processing step than an actual detection step, especially for more intricate shapes for which segmentation cannot be guaranteed. Compared to background subtraction, however, segmentation has the additional difficulty that one needs to determine which subset of segments (from all detected segments) are the ones which cover the shape to track. As such, while technically more flexible, segmentation may actually pose too complex challenges for our object tracking context.

To conclude this section, let us mention our completely different approach to segmentation which is proposed in chapter 4. In contrast to all other methods, we use object representation techniques (Sec. 2.1.1), specifically medial axes, to both model *and* segment shapes from the surrounding background [314]. As we shall show there, this allows a good control of the segmentation level-of-detail with minimal user intervention.

2.2.3 *Feature Detection*

Where the detection methods described earlier try to divide the image in segments and then search for a segment containing the object of interest, feature detection methods try to find features corresponding to the object of interest in the image directly. While there are a lot of different categories of feature detection methods, we will focus on the two categories most appropriate for our use case: *point detectors* and *template matching*.

Point detectors: While images are entirely made up of points (pixels), point detectors try to find the points that stand out from their surrounding, the so called *interest points* or feature points. For example, these can be points for which the image intensity changes sharply compared to their neighbors. Invariance to illumination and camera viewpoint are qualities of most interesting points which makes them suited for use in tracking applications.

A more in-depth overview on interest-point detectors can be found in [226]. Next, we will give an overview of some of the more important methods.

A commonly used method is Moravec's interest operator which looks for the maximum intensity change in an image patch over different directions (horizontal, vertical, and (anti-) diagonal) [183].

The Harris detector is slighty more involved, since it is based on a matrix of first order derivatives [102]. The interest points are determined based on the trace and determinant of this matrix of first order derivatives, whereas the interest point detection of the KLT tracking method uses the eigenvalues of this matrix [238].

The interest point detection methods mentioned above are all (theoretically) invariant to rotation and translation, but they are not invariant to changes in projection or affine transformations. As a solution, the scale invariant feature transform (commonly referred to as SIFT) was proposed [160]. The SIFT method creates description vectors for interest points over a scale space representation of the image, resulting in high-dimensional vectors representing the interest points. When trying to locate a given object in an image, the same technique is applied to the target image to find the representing vectors of feature points in the images. Following that, the description vectors of the target object are matched to the description vectors in the image to try and locate the target object.

Various ways have been proposed to speed up SIFT, such as Speeded-Up Robust Features (SURF) which improves the speed by reducing the complexity of the description vector combined with a different matching technique [19]. A large gain can be made by increasing the performance of the matching technique, since the target object will be matched against a range of images to try and locate it. Secondly, the reduction of complexity of the description vector means that all steps will perform better, since the description vector is essential throughout the entire detection pipeline.

Template matching: In contrast with the previous detection methods, which identify interesting points or segments of the image, template matching techniques try to directly find the location of an example image (the template) in the image. As discussed in subsection 2.1.2, the template image captures both the shape as well as the appearance of the target object. At a high level, template matching methods can be seen to generalize interest-point detectors by generalizing the concept of a point to that of a (small) template image. This gives more flexibility in defining what an interesting feature is. The result of applying template matching to an image is typically a map representing how well the template matched while moving over the entire image [18]. A drawback of also capturing object appearance in the template image is that traditional (cross correlation based) template matching methods are quite sensitive to differences in intensity between the input image and the template image. The Normalized Cross Correlation (NCC) matching technique tries to solve the problem of intensity difference between the template and input image. The computation of the NCC match coefficient can be sped up by performing the necessary convolutions in the Fourier domain [151], making it possible to perform real-time detection. Faster solutions than the Fourier approach have also been proposed [29].

2.3 OBJECT TRACKING

The goal of object tracking is to find the trajectory of an object given a sequence of images. This can be seen as adding a temporal constraint to the spatial constraints of the detection methods described above. In this section, we will first give a general introduction to tracking methods and possible temporal constraints, followed by a more in-depth look at some selected tracking methods appropriate for our intended use-case of tracking cow teats.

When determining the trajectory of an object, the most important, but also most challenging, task is that of determining the object correspondence

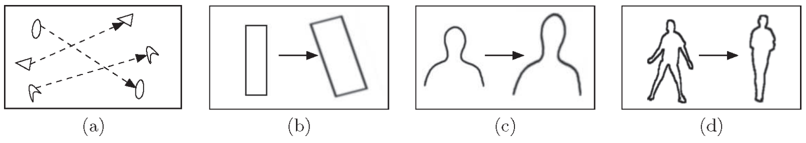


Figure 2.5: Object correspondence for different object representations, from [306]: (a) Multipoint correspondence, (b) parametric transformation of a geometric primitive, (c, d) contour evolution.

between frames. Different object representations (see subsection 2.1.1) lead to different object correspondence methods and therefor different categories of tracking methods, as can be seen in Figure 2.5. Most tracking methods require an external detection method to find the objects in the image. However, some tracking methods are capable of doing detection and tracking jointly. The latter category usually still requires external (human) input to indicate the object to be tracked at the start of the tracking process.

In their overview paper on tracking methods, Yilmaz et al. [306] give a list of constraints for point tracking most of which are, in our opinion, applicable to the broader problem of object tracking. In the following, we provide a summary of these constraints.

PROXIMITY The most simple constraint to set on object motion is based on the assumption an object will not move a lot between frames. Therefore, the object in the new frame that is closest to where the object was in the previous frame is most likely the same object (see Figure 2.6(a)).

MAXIMUM VELOCITY By assuming a maximum velocity at which the object can move, we can define a spherical neighborhood with the object position in the previous frame as the center, constraining the possible location of the object in the current frame within this neighborhood (see Figure 2.6(b)).

SMALL VELOCITY CHANGE Besides assuming a maximum velocity, we can also constrain the possible changes in velocity. That is, we can assume the tracked object will keep moving more or less in the same direction with the same speed, so no jumps in direction and/or speed will occur (see Figure 2.6(c)). This essentially captures a smoothness constrain on the trajectories of the moving object.

COMMON MOTION When using multiple points to represent an object (either as separate points or points on the silhouette) we can assume that points which are close to each other move in the same way (see Figure 2.6(d)). This essentially captures a rigidity constraint on the tracked object.

RIGIDITY For rigid objects, we can assume all points on the object will move in the same way, such that the distance between all points are

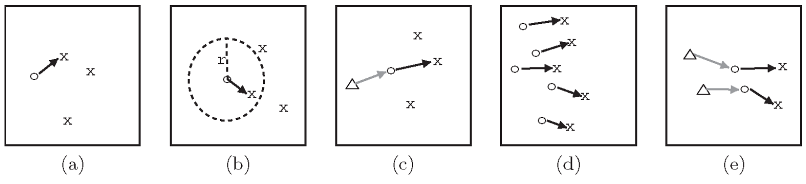


Figure 2.6: Tracking constraints, from [306]: (a) Proximity, (b) maximum velocity (with r the radius of the spherical neighborhood), (c) small velocity change, (d) common motion, (e) rigidity. A cross ($\times$) indicates object position at frame t; a circle ($\circ$) indicates the position at frame $t-1$ and, where applicable, a triangle ($\triangle$) indicates the position at frame $t-2$.

constant (see Figure 2.6(e)). This is another formulation of the aforementioned rigidity constraint.

The constraints above can be combined to further specify the restrictions to apply to the motion of the tracked object, for example, combining the proximity and small velocity change constraints leads to a smaller range of the possible locations for the object in the new frame than each of the individual constraints when taken separately. Using appropriate methods to evaluate these constraints, we can construct a method to keep track of objects over a sequence of images.

While the point correspondence problem that point based tracking methods have to solve is complicated, the algorithms themselves are easier to explain. One of the earliest methods for solving the point correspondence problem is that of Sethi and Jain [233]. Their approach is based on the constraints for proximity and rigidity and uses an iterative optimization algorithm to find the trajectories for all points available in the starting image. However, the approach does not handle the (dis)appearing of points or possible occlusions. Salari and Sethi [224] propose a solution to this problem by adding placeholder points where points are expected to be, but are not found during point detection.

Another approach to object tracking is to create a statistical model of the object (or objects) to track, with the model state representing the object's position, velocity, and acceleration. By using statistical correspondence instead of direct correspondence, we can account for uncertainties in the model and noise that is present in the input sequence.

Probably the most well-known object state estimation method is the Kalman filter which gives the optimal state estimate when the noise and model state are assumed to have a Gaussian distribution. The Kalman filter uses two steps to determine the new state of an object, the prediction and correction. During the prediction step, an estimate of the new state is made based on previous observation and the model. The subsequent correction step updates the current object state taking into account both the estimated new state as well as the current observation.

The Kalman filter has been applied for tracking purposes [17], for example, to estimate point trajectories in noisy images [30]. Another example of estimating point trajectories using a Kalman filter is the work of Rosales and Sclaroff [221], where they are used to determine the 3-dimensional trajectories of the tracked object based on 2-dimensional images.

3 RELATED WORK: VISUAL ANALYTICS FOR MULTIDIMENSIONAL TRAILS

Visual analysis of large, multidimensional, and time-dependent data is one of the key focuses of information visualization and visual analytics. As this problem is central to our research questions, most notably *RQ*2 (Sec. 1.4), we present here an overview of existing techniques that address this task. We focus on the relative pro's and con's of such techniques, with an eye on our specific problem contexts, as overviewed in Chapter 1.

3.1 CONTEXT

Simply put, the context of visually analyzing multidimensional time-dependent data can be stated as: How can we present (large amounts of) such data, so that users can effectively and efficiently accomplish a number of tasks? Such tasks cover the identification of outliers (data items that significantly differ from the main corpus of observations); finding which specific dimensions of the recorded data make certain observations different; and emphasizing the dynamics of the recorded data, or how measured properties change over time.

To address the above, we need first to provide a (formal) notation of our data of interest is; and next, describe the tasks of interest in terms of this notation. These items are detailed below.

3.1.1 *Multidimensional Data*

Generally, data in our problem context can be modeled as a set of *observations* having several measurable (and measured) *dimensions* [185, 271]. The easiest (and typically, most frequent) way to describe such data is a bottom-up approach, starting from the lowest-level (simplest) data items, and subsequently proceeding to group, or aggregate, such items based on the most relevant aspects (dimensions or measurements) they share.

At the lowest level, we identify an *observation* as being a tuple

$$\mathbf{x}_i = (x_i^1, \ldots, x_i^D). \tag{3.1}$$

Here, the elements x_i^j, $1 \leq j \leq D$ are the so-called dimensions of observation i. Let us denote next the collection, or set, of all such observations $\mathbf{i}_i$ by DS, *i.e.*, $DS = \{\mathbf{x}_i\}$. The observations $\mathbf{x}_i$ are individually measured aspects of a certain phenomenon, such as, for example, positions, heights, speeds, heading directions, and identifiers of airplanes moving over a given region of space and a given time period [112, 118, 119, 137]. That is, each measurement on a given airplane at a given time moment generates a different observation $\mathbf{x}_i$. For notation simplification, we next denote all values of dimension j over our measurements by $\mathbf{x}^j$. In our context, one of the dimensions j denotes time, *i.e.*, for that respective dimension-index j, $\mathbf{x}^j \subset \mathbb{R}^+$.

The collection $\{\mathbf{x}_i\}$ of all such measurements denotes, thus, a multidimensional, time-dependent, dataset. The dimensionality of such a dataset is equal to the number of independent measurements, or D. The number of measured entities, *i.e.* $|\{\mathbf{x}_i\}|$, is next denoted by N, *i.e.*, $1 \leq i \leq N$.

The dimensions of our measured data can have several properties. Such dimensions are also denoted in information visualization by the name of variables, attributes, components, features, or columns. Following [185, 271], among others, we can classify such dimensions by the *properties* they have, or more explicitly, the operations they allow on their values. A well-accepted classification is as follows [45, 170, 185, 271].

- **Quantitative:** Quantitative attributes, support operations comprising addition, subtraction, and multiplication by a real number. In simple terms, such attributes are real values, *i.e.* $R \subset \mathbb{R}$. Such attributes are commonly found in computer vision and in so-called *scientific visualization* (scivis) applications, such as the exploration of geographical data, computational flow dynamics (vector fields), numerical simulations of 2D or 3D physical problems, or exploration of body scans in medical science (*e.g.* CT and MRI scans) [101]. Arguably the most important property of such attributes is that they allow *interpolation.* Briefly put, this allows us to (a) estimate the value of such an atribute x^j, based on recorded samples, at a spatial location different (but close to) the locations where the samples were taken. Interpolation is crucial for operations such as producing (typically piecewise-continuous) reconstructions of the signal x^j from its samples; eliminating noise during this reconstruction; detecting outliers; resampling, *i.e.*, producing a set of measurements $\{x_i^{j\prime}\}$ from a given set of measurements $\{x_i^j\}$ so that the cardinalities $\|\{x_i^{j\prime}\}\|$ and $\|\{x_i^j\}\|$ are different; and, last but not least, reducing the cardinality N of the set of observations by means of aggregation, an operation key to handling large datasets.

- **Integral:** Integral attributes, sometimes also called discrete attributes, allow only operations such as addition and subtraction (they do not allow weighting with a real-valued parameter). Such attributes are typically a subset of $\mathbb{Z}$. Such attributes are very frequently met in information visualization (infovis). Examples hereof are *counts* of various quantities, such as numbers of persons, transactions, sold/bought items, or similar [185, 264, 286]. A key difference with quantitative attributes is that integral (discrete) attributes do *not* allow intepolation.

- **Ordinal:** Ordinal attributes allow ordering, *i.e.* define the relations $<$, $>$, and $=$ over their range. Examples of such attributes are ordered sequences of measurements, such as *e.g.* Likert 5-point scales used to assess the quality of an item, or ranks in an organizational hierarchy (*e.g.* rector, dean, full professor, associate professor, assistant professor, postdoc, PhD student, MSc student, BSc student in an university organization). Ordinal attributes do not allow interpolation or direct summarization via aggregation – in other words, we cannot say what it means to be between a PhD student and a postdoc, or what the 'average' of a PhD student and a postdoc would be.

- **Categorical:** Categorical, also called nominal, attributes allow only the comparison operation – that is, they only allow telling if two categorical values are the same or different. Examples of categorical attributes are types of elements in a collection, such as brands of car vehicles; gender (male or female); of file types in an operating system (*e.g.* executables, text documents, database, source code, and others). Categorical attributes are generally the most complex ones to analyze and visualize [185, 271]. Indeed, as they do not allow interpolation (thus, aggregation), it is very hard to summarize large sets having many categorical attribute-values in a compact manner.

- **Text:** Text attributes are, formally speaking, identical to categorical attributes – that is, in the conservative case, any two text strings a and b can be seen as being either identical, or different. However, in practice, differences exist: For text, we can (usually) compute a syntactic or semantic difference, or distance, between any two text strings, which brings such attributes in the realm of quantitative attributes [150, 201, 204, 205]. As compared to all other attribute types we are aware of, the analysis (and/or usage) of text attributes is strongly influenced by their semantics, with two possible outcomes: Either one knows how to compute a continuous *similarity* between text attribute values, in which case such attributes get largely handled as quantitiative attributes; or such a similarity cannot be computed, in which case such attribute values are treated as individual tokens, thus, just as categorical ones.

- **Relational:** Relational attributes essentially indicate subsets of observations, in a given dataset, which share some common properties. As such, their focus is not the domain (range) values of individual attributes $\mathbf{x}^j$, but relations between dimensions $(j, j') \in D \times D$, or, in the more complex case, subranges of such dimensions. In simple terms, relations attributes can be seen as modeling graphs where nodes are individual observations $\mathbf{x}_i$ and edges are problem-specific associations between such observations $(\mathbf{x}_i, \mathbf{x}_j)$. As such, relational attributes significantly differ from all other attribute types discussed earlier, in the sense that they do not characterize *individual* observations, but *groups* of observations. Relational datasets are often met in information visualization and are the focus of graph and network visualization [59, 61, 62, 67].

3.1.2 *Trail Data*

Within the context outlined in the previous section, a *trail* is defined as a subset of observations $\mathbf{x}_i$ that relate to consecutive (time-wise) measurements or the attributes of a single *entity*. Thus, to formally define a trail, we need two things:

- **Time:** We need to specify one attribute $1 \leq j \leq D$ of our dataset which represents time. That is, two observations $\mathbf{x}_{i1}$ and $\mathbf{x}_{i2}$ for which the values x^j_{i1} and x^j_{i2} of a given dimension j can be ordered, *i.e.* $x^j_{i1} < x^j_{i2}$, are said to be measured at two consecutive time moments x^j_{i1} and x^j_{i2}. Let us denote the dimension j describing time by j^{time}.
- **Identity:** In general, observations in a dataset do not describe independent and unrelated entities. For instance, imagine a table where rows $\mathbf{x}_i$ record positions of S moving 3D shapes over time. Clearly, several such rows describe the position of the *same* shape over time (S in total). Thus, it makes full sense to study the entries $\mathbf{x}_i$ grouped by the shape (object) they describe. Simply put, it does not (generally) make sense to lump the positions of a shape with those of another shape. To do this, one does (in general) assign a specific attribute $\mathbf{x}^j$ to denote the *identity* of an object of interest. That is, the set of observations $\{\mathbf{x}_i | x^j_i = id\}$ denotes all information for a given shape *id* [264, 286]. Let us denote the dimension describing identity by j^{id}.

Given the above, we now can define a *trail* as being the ordered set of time-sampled observations, in a given dataset, that pertain to a single shape:

$$\begin{aligned} T_k = \{\mathbf{x}_i \in DS | \forall \mathbf{x}_u \in T_k, \mathbf{x}_v \in T_k, \exists j^{id} \in \{1, \ldots, D\} :\\ x_u^{j^{id}} = x_v^{j^{id}} = k \ \wedge \\ \exists t \in \{1, \ldots, D\} : x^t_u < x^t_v, \forall (u, v) \in D \times D\}. \end{aligned} \tag{3.2}$$

In simpler words, a trail T_k is a set of observations $\mathbf{x}_i$ in our given dataset DS which (a) belong to the evolution of the same shape k, and (b) have a 'time' attribute $x^t \in \mathbb{R}^+$ that tells when they were observed. Trails can depict the change of position of a physical entity over Euclidean space, such as the position of a vehicle or another tracked object over time [112]. However, in the general case, they can represent the change of attributes of a data entity in any data space, such as the learning dynamics of the values of neurons in a deep artificial neural network [212]. In our work, we are interested by both types of trails.

Trail data shares, obviously, commonalities with both multidimensional data (as many observation attributes can be sampled during the observation's evolution), dynamic data (as there are observations having explicit identities which change in time), and spatial data (as, in most cases, the recorded observations evolve in a 2D or 3D space). Given the above, trail visualization has affinities with multidimensional visualization, dynamic visualization, and spatial visualization. We outline these affinities next.

3.2 MULTIDIMENSIONAL DATA VISUALIZATION

The general focus of multidimensional data visualization is to show a dataset *DS* having high-dimensional observations $\mathbf{x}_i$ (Eqn. 3.1) so that one can reason about both observations and their dimensions.

Many visualization methods exist in this class. To compare them, we need to group them in some way. One possible such grouping is based on the emphasis of the methods on *observations* or *dimensions* – that is, which of the two aspects is more in focus for the method at hand. Given this, we classify such visualization methods into dimension-centric (Sec. 3.2.1) and observation-centric (Sec. 3.2.2). Methods in both classes are overviewed next.

We note that other taxonomies of multidimensional visualization methods exist, such as the one proposed by Chan [41], which revolves around the type of visual encoding being used (geometric, icon-based, pixel-based, hierarchical, graph-based, and hybrid); or the one proposed recently by Coimbra [45], which identifies axis-based methods, space-filling methods, projections, and other approaches. We argue that our taxonomy, which is data-centric, is more appropriate in our context.

3.2.1 *Dimension-centric methods*

Dimension-centric visualization methods focus on supporting exploration tasks where the main questions of the user revolve around the *dimensions* of a dataset. To understand this, consider that our multidimensional dataset *DS* can be thought of as a table, where observations $\mathbf{x}_i$ are rows, and dimensions $\mathbf{x}^j$ are columns, respectively. Dimension-centric visualizations, thus, focus mainly on answering questions concerning the columns of such a table. Example questions are: Which are the extreme values of a column? Which columns of a table show similar variations, *i.e.*, are strongly correlated? Which columns are inversely correlated? Which columns are independently varying? Which columns show high, respectively low, variance?

Several dimension-centric methods exist for visualizing multidimensional data, as follows.

3.2.1.1 *Tables*

Arguably the simplest, and oldest, way to visualize a multidimensional dataset is to directly follow the table metaphor: Observations $\mathbf{x}_i$ are drawn as textual rows, so that their values x_i^j corresponding to a given dimension j get aligned vertically to yield columns $\mathbf{x}^j$. Table views allow detailed investigation of the data values x_i^j, and can be enhanced by showing these values by mapping these to *e.g.* scaled and/or colored bars drawn in the background of the cells (Fig. 3.1a). However, while good for examining details, table views cannot handle more data than a few tens of rows (observations) and 10 up to 20 columns (dimensions).

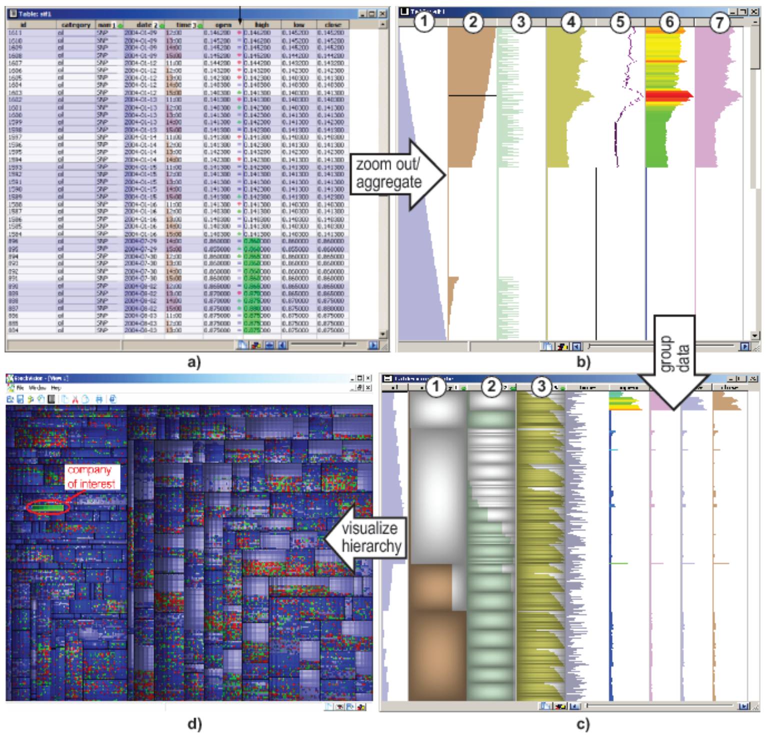


Figure 3.1: Table visualization. (a) Classical layout; (b) table lens; (c) grouping data; (d) treemap constructed from (c). Visualizations created with the TableVision tool [264]. See Secs. 3.2.1.1 and 3.2.1.2.

A simple but effective way to increase scalability of table views and also support dimension-centric tasks such as comparing entire columns is provided by the table lens technique [210]: Inuitively put, every table row is drawn as a horizontal pixel line, where individual cells are rendered as scaled and/or colored bars that map the values x_i^j, as if 'zooming out' the classical table view. Figure 3.1b shows a table lens image for the data in Fig. 3.1a. It is now easy to notice that, for instance, columns 1 and 2 are inversely correlated; columns 4 . . . 7 are strongly directly correlated; and column 3 does not have a clear relationship with any of the remaining columns. Table lens techniques are, observation-wise, very scalable – they can easily show hundreds of thousands of observations (rows) on a single screen having just a few thousand pixel lines in the vertical dimension, by using suitable data aggregation and clustering techniques [264].

3.2.1.2 *Generalized treemaps*

One issue of table views (or table lenses) is that, while they allow one to compare *entire* dimensions, they are not very effective in helping one reasoning about specific *ranges* of dimensions. For instance, consider two dimensions $\mathbf{x}^j$ and $\mathbf{x}^k$: While there may not be any apparent correlation for the entire ranges of $\mathbf{x}^j$ and $\mathbf{x}^k$, observations that have low values for $\mathbf{x}^j$ always have high values for $\mathbf{x}^k$.

Reasoning about ranges of dimensions is supported by group-and-sort techniques that can be directly applied on table views. Figure 3.1c shows an example: Here, we first group all observations (rows) based on identical values of column 1. This partitions the data into four groups, as there are four different values for $\mathbf{x}^1$; such groups can be visually emphasized by overlaying them with shaded cushions, as shown in the figure. Next, for each such group, we repeat the grouping process on column 2, and subsequently on column 3. As a result, the entire dataset is transformed into a *hierarchy*. Here, the root represents the entire dataset; each subsequent level represents the grouping of observations based on the values of a different dimension $\mathbf{x}^j$. The tree leaves represent the individual observations $\mathbf{x}_i$. The resulting tree can be next visualized using, for instance, treemap techniques [283], with observations colored on the value of one dimension and scaled by the value of another dimension.

Depending on the order in which columns are considered for grouping, and the exact type of grouping (*e.g.* based on exact equality of values or on user-defined data ranges), an entire family of so-called generalized treemaps can be constructed from a single tabular dataset. This allows exploring the observations based on their similarity given by a set of user-chosen dimensions. This technique has been used in many applications for visualizing, for instance, business and finance data [86, 286]. However, table views and table lenses are, in our view, not optimally suited for handling *spatial* data, such as emerging from the context of computer vision or tracking applications, as they do not intuitively show how the values of dimensions (samples) exist, or evolve, in 3D space.

3.2.1.3 *Scatterplots*

Generalized treemaps are very effective when the tasks at hand revolve around analyzing observations *grouped* by various criteria. Besides showing the similarity of observations by spatial grouping, treemaps can also visually encode up to typically two dimensions per observation, shown via color, respectively size. However, seeing correlations (or the lack thereof) of these dimensions can be hard, as a treemap does not usually provide a 'reading order' to scan these two dimensions from their low to their high values.

Scatterplots are probably the best-known tool that help with the above task of detecting and analyzing correlations. Observations are projected as points into a low (typically two- or three-dimensional) space, whose axes encode the values of two, respectively three dimensions $\mathbf{x}^j$ of the dataset. An additional dimension can be mapped to point colors. This allows one to easily see the range of values of these dimensions; detect outlier observations; globally perceive the distribution (spread) of data points over the dimensions' ranges; and detect potential correlations of the displayed dimensions.

Scatterplots are highly scalable in the number of observations N. Overplotting can occur when tens of thousands of observations (or more) are drawn over a small screen area. However, in such cases one can visually encode the spatial density of observations by using techniques such as transparency, alpha blending, and accumulation maps. Additionally, observations can be drawn ordered by the value of a given dimension of interest, so that extremal values show up better [120]. When color is used to map an additional dimension, care should however be used as blending and color-coding are, in general, not commutative operations (see *e.g.* [271], Sec. 5.1).

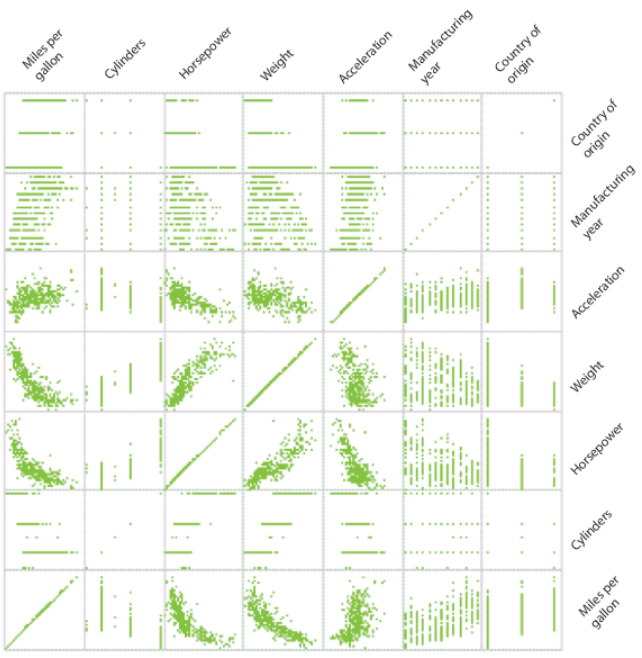


Figure 3.2: Scatterplot matrix (SPLOM) showing a dataset of cars having $D = 7$ dimensions. See Sec. 3.2.1.4.

3.2.1.4 *Scatterplot matrices (SPLOMs)*

As outlined above, scatterplots are limited to showing around three independent dimensions. Three-dimensional scatterplots can add a fourth dimension; however, they suffer from problems such as occlusion and the difficulty to choose a suitable viewpoint [230]. While interactive methods have been proposed to alleviate such issues, three-dimensional color-coded scatterplots are still challenging to use [44].

For high-dimensional datasets (D in the range of tens or even hundreds), the basic scatterplot metaphor can be adapted in several ways. First, one can compute scatterplots of all pairs of dimensions, and display then arranged in a matrix, where each row, respectively column, groups plots for the same dimension $\mathbf{x}^j$. For a dataset of D dimensions, we thus need $D(D+1)/2$ plots, as the matrix is symmetric. The resulting configuration is called a scatterplot matrix, or SPLOM. SPLOMs are an instance of the so-called *small multiple* design metaphor, which essentially shows a small number of visualizations side-by-side, all having the same visual encoding, but depicting different parts of the data [20, 279].

Figure 3.2 shows a SPLOM for a dataset where observations are cars, each having $D = 7$ attributes of various types (*miles per gallon*, *weight*, *acceleration*: quantitative; *year of fabrication number of cylinders*: ordinal; and *country of origin*: categorical). As visible, correlations such as the inverse one between *miles per gallon* and *weight* are easy to spot. Moreover, the diagonal cells show the distribution of values of each dimension, much like a one-dimensional histogram.

While effectively increasing the number of dimensions that can be visualized in the same time, SPLOMs have several problems. First, they cannot accommodate more than roughly $D = 10$ dimensions, as the matrix size increases quadratically with D. Secondly, they require a non-negligible amount of 'visual scanning' to analyze all possible interesting patterns in the data. Thirdly, they do not support observation-centric tasks – indeed, an observation is essentially a set of $D(D+1)/2$ points, one per matrix cell. While observation-centric analyses can be made easier by interactive brushing and linking between cells, overplotting can diminish the effectiveness of such explorations. Finally, SPLOMs are less effective in depicting phenomena that involve more than *pairs* of dimensions, and as such have also been called 'multiple *bivariate* visualizations' [100].

The problem of dimensional scalability of SPLOMs can be alleviated in several ways. First and foremost, one can automatically analyze all D^2 dimension-pairs to detect interesting patterns, such as pairs of strongly correlated (or, conversely, independent) dimensions, or clusters of observations. These can then be presented as a (typically small) sequence of scatterplots, ranked in terms of interestingness. The class of techniques achieving this is known globally as *scagnostics* [146, 280, 293]. Separately, one can use interactivity to generate a rich set of 2D scatterplots by allowing users to select the dimensions of interest to map to the visual x and y scatterplot axes. Interpolation can be used to create smooth transitions as these mappings change, thereby helping the user to maintain the so-called 'mental map', *i.e.*, follow patterns created by observations between views [74, 120]. Interestingly for our context, such techniques have also been used to explore multivariate trails of aircraft [112, 115].

3.2.1.5 *Parallel coordinate plots (PCPs)*

The challenge of navigating between an observation-centric view and a dimension-centric view on data, as well as the scalability issues of SPLOMs, have led to the development of parallel coordinate plots (PCPs) [125]. An

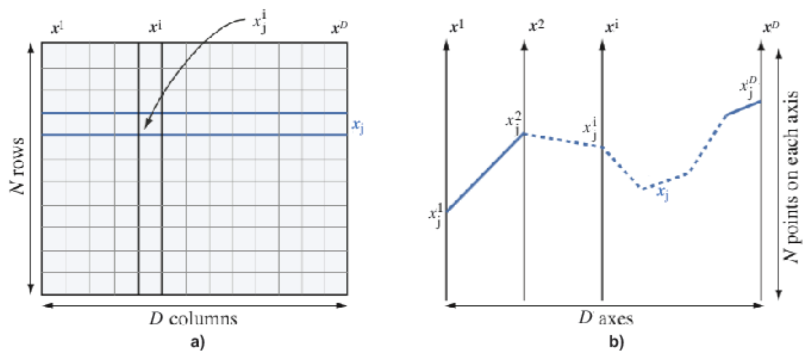


Figure 3.3: Relationship of table views (a) and parallel coordinate plots (Sec. 3.2.1.5). Image from [271], Sec. 11.5.1.

easy way to explain PCPs is by contrasting them to table views (see Fig. 3.3). In a table view, each dimension j is essentially a vertical column listing the values x_i^j over all observations i; and each observation i is a row listing the values x_i^j over all dimensions j. In a PCP, each dimension is a vertical axis covering the range of $\mathbf{x}^j$; and each observation is a polyline that links one point on each axis j representing the value of x_i^j. In a table view, data can be thought of as being sorted in order of *observations*; in contrast, in a PCP data is sorted in order of the actual *values*. In both cases, *dimensions* can be freely sorted by the user along the horizontal axis.

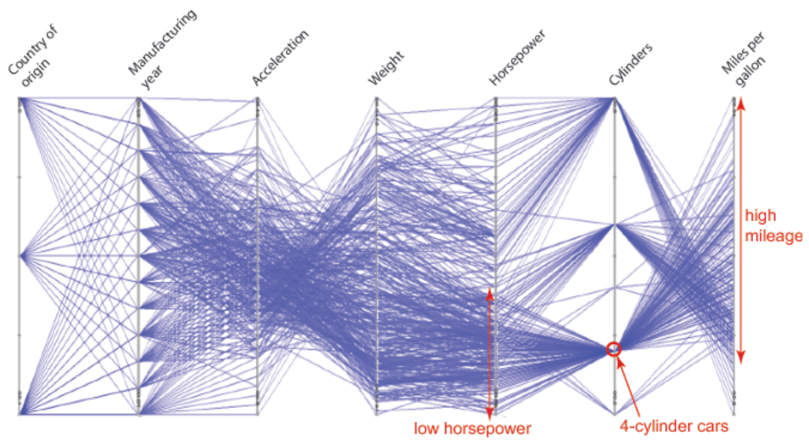


Figure 3.4: Parallel coordinate plot (PCP) for the same car dataset as shown in Fig. 3.2. See Sec. 3.2.1.5.

Figure 3.4 shows a PCP for the car dataset shown earlier in Fig. 3.2. In contrast to SPLOMs, it is now easier to detect patterns involving more than two dimensions, by visually following bundles of closely-spaced lines (sets of observations). For instance, by following the line bundle passing through the marked red dot, we can see that most four-cylinder cars have a high mileage and a relatively low horsepower. The distribution of values along each dimension is visible in much the same way the diagonal cells of a SPLOM show, and similar blending and accumulation mechanisms can be used to alleviate clutter problems. Direct and inverse correlation patterns show up as groups of parallel, respectively crossing, lines.

In contrast to SPLOMs, PCPs scale better with the number D of dimensions – more precisely, linearly rather than quadratically. However, visual clutter is now more problematic, as there is more ink used per observation – one line for PCPs instead of a set of D^2 points for SPLOMs. More problematically, relating dimensions whose axes lie far away in the PCP is difficult. While interactive brushing and linking can alleviate such issues, just as for SPLOMs, PCPs scale less well with the number N of observations than SPLOMs. Additionally, the ordering of axes critically influences the ability to see patterns in the data – an issue that SPLOMs do not have. A final challenge of PCPs is that they require more time to learn, and they are less intuitive, we argue, than all other visualization metaphors discussed so far in Sec. 3.2.1. Also, similar to table views, PCPs do not intuitively map the spatial nature of (3D) data, such as relative positions and motion directions.

3.2.2 *Observation-centric methods*

As outlined in the beginning of Sec. 3.2, observation-centric methods pose the focus of the visualization on the actual data items, or observations, rather than on their (many) dimensions. As such, these methods are less suited to reason about *e.g.* correlations of dimensions or distributions of data values. However, they are better at supporting tasks that revolve around identifying groups of *similar* observations or *outlier* observations.

3.2.2.1 *Dimensionality reduction methods*

Given the above tasks, similarity is at the core of many observation-centric methods. A particular class of such methods is formed by so-called *dimensionality-reduction* (DR) techniques. Formally put, a DR technique can be seen as a function

$$f : \mathbb{R}^D \rightarrow \mathbb{R}^d, \tag{3.3}$$

where $d \ll D$ (typically, $d \in \{2, 3\}$). The underlying idea of f is to transform a given a dataset $DS \subset \mathbb{R}^D$ into a low-dimensional dataset $DS' = \{f(\mathbf{x}_i) | \mathbf{x}_i \in DS\}$, $DS' \subset \mathbb{R}^d$, so that the so-called 'structure of the data' is preserved [168, 263]. This structure is defined by means of similarity – or more precisely, variation thereof – over a given dataset DS. To understand this, consider *e.g.* some distance metric $\delta : DS \times DS \rightarrow \mathbb{R}^+$. The structure of a dataset DS can be seen in terms of distribution of distance values $\{\delta(\mathbf{x}_i, \mathbf{x}_j) | \forall (\mathbf{x}_i, \mathbf{x}_j) \in DS \times DS\}$. For instance, a dataset DS composed of two compact, but well-separated similar-size observation clusters in $\mathbb{R}^D$, will show a bimodal balanced distance distribution, whereas a dataset featuring a single compact cluster will show a unimodal distribution. Another way to express data structure is to look at the set of k-nearest neighbors of each observation $\mathbf{x}_i \in DS$ [13, 148].

DR methods have several key advantages compared to all multidimensional visualization techniques discussed earlier:

- *Scalability:* DR methods are highly scalable *both* in the number of observations N and number of dimensions D. One can argue that they are close to the optimum – indeed, we require just one pixel to draw an observation, and the required screen space by a DR method is independent on D. No other multidimensional visualization method we are aware of can achieve both these constraints;
- *Familiarity:* DR methods essentially reuse the scatterplot metaphor (Sec. 3.2.1.3) which, as we have seen, is one of the most familiar ones in the multidimensional visualization area;
- *Observation-centric:* As already said, DR methods focus on observations, not dimensions. As such, analysis tasks where one is interested in finding how observations relate to each other are well served by such methods.

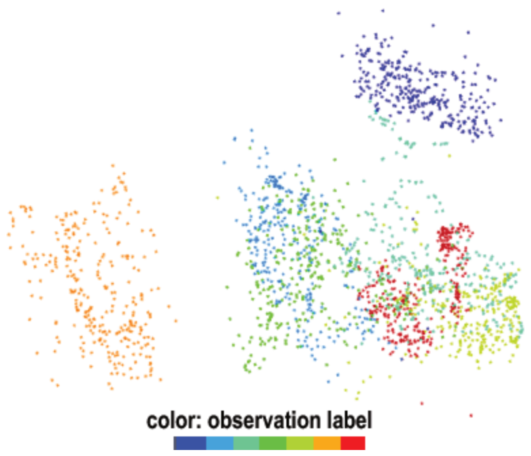


Figure 3.5: Dimensionality-reduction visualization for a dataset of image fragments used in the context of classifier design. Image taken from [44].

Figure 3.5 shows the result of applying a DR technique [130] to a set of 2300 observations for the task of image classification. Each observation is a small pixel block pulled from seven natural-scene images, each such image being of a different type. For each such block, 19 dimensions are computed, including typical features used in the design of image classifiers [300]. An additional categorical attribute is manually added by a human observer, indicating the class (type) of the image each block has been taken from. This attribute is color-coded in Figure 3.5. Based on this image, one can assess how well the measured attributes correlate with the class attribute, and thus how good these are for the potential task of designing a classification system able to infer the class attribute. Examples of projections being used to this end are given in [213].

DR methods will play an important role in our visual exploration of high-dimensional time-dependent data (Chapter 6). Hence, we provide next a brief overview of such methods.

3.2.2.2 *Types of DR methods*

Dimensionality reduction methods are well known in statistics and data science since over a century, when Principal Component Analysis (PCA) was introduced [131]. Tens of DR methods have been proposed In the last decade, as recent surveys in the area are showing [164, 249]. However, the (often subtle) differences in aims, constraints, and scope of such methods are not always clear – owing, probably, to the fact that DR techniques are at the crossroads of several traditionally separated disciplines such as statistics, machine learning, data science, and data visualization.

To better understand the above-mentioned differences, we propose next four axes along which DR methods can be classified.

Preservation aims: Key to the idea of preservation of data structure is that the characteristics of interest are similar over $\mathbb{R}^D$ and $\mathbb{R}^d$. Depending on how data structure is quantified, we have two main classes of DR methods.

Distance-preservation DR methods aim to preserve the relative distances between point-pairs in DS and DS'. In other words, they aim to have the ratio $\delta(\mathbf{x}_i, \mathbf{x}_j)/\|(f(\mathbf{x}_i), f(\mathbf{x}_j)\|$ relatively constant for all point-pairs $((\mathbf{x}_i, (\mathbf{x}_j) \in DS \times DS$. Here, $\| \cdot \|$ denotes Euclidean distance over $\mathbb{R}^d$. Distance preservation is typically quantified by computing the so-called aggregate normalized stress (*cf e.g.* [130])

$$\sigma = \frac{\sum_{(\mathbf{x}_i, \mathbf{x}_j) \in DS \times DS} (\delta(\mathbf{x}_i, \mathbf{x}_j) - \|f(\mathbf{x}_i), f(\mathbf{x}_j)\|)^2}{\sum_{(\mathbf{x}_i, \mathbf{x}_j) \in DS \times DS} \delta(\mathbf{x}_i, \mathbf{x}_j)^2} \tag{3.4}$$

Variations of this metric are proposed by Martins *et al.* to find specific point-pairs which are mapped too close, or too far, by the DR function f [168]. Most known DR techniques fall into the class of distance-preservations methods, *e.g.* [83, 124, 130, 193, 199–202, 246, 247].

As outlined at the beginning of Sec. 3.2.2.1, another way to encode data structure, apart from looking at point-pair distances, is to consider the neighborhoods of points. In this metaphor, good DR methods are those which preserve the k-nearest neighbors of points between DS and DS'. Neighborhood preservation is often more useful than 'pure' distance preservation, for instance in cases where one is interested to reason about clusters (groups) of observations. In such cases, one may even want to bias distances in DS', as compared to DS, so that observation groups appear more compact and better separated from each other. Note that, in the ideal case, a perfect distance-preserving DR method is also perfectly preserving neighbors, for any $1 \leq k \leq N$. However, the converse is not necessarily true.

Neighborhood-preservation DR methods are relatively newer, and less well-known, than distance-preservation DR methods. Examples include LoCH [80] and the by now famous t-Stochastic Neighbor Embedding (t-SNE) [162, 163], which we will also use later on in Chapter 6, given its good ability to separate groups (clusters) of similar observations. Quantifying the quality of such methods is done by *e.g.* computing metrics such as the neighborhood hit [202], silhouette coefficient [260], and various types of local set-based metrics that compare the k-nearest neighbors of a point in *DS* and *DS′* [169, 200]. Variants of such metrics consider group-membership preservation, *i.e.*, whether an observation is seen to be part of the same (compact) group of observations [168]. While assessing projection quality is important, we consider this to be out of the scope of our work.

Distance-based *vs* dimension-based: A second way to classify DR methods is to look at the type of information they use to quantify similarity of observations. Two situations exist here. First, one can design the function f (Eqn. 3.3) to directly use the observations $\mathbf{x}_i \in \mathbb{R}^D$. One advantage hereof is that only the actual dataset *DS* to be mapped to d dimensions is needed – no additional data structures need to be computed and/or stored. Another advantage is that the obtained low-dimensional dataset *DS′* can be next explained by showing how the original high-dimensional attributes $\mathbf{x}^j$ influence the point placement, therefore making the low-dimensional image more usable and useful [44, 168, 243, 245]. DR methods that explicitly use the high-dimensional attributes are also known as *projections*.

However, there are cases when one does not avail of *explicit* dimensions to describe the observations' similarities, but however has this similarity information. An example is the situation when we have a dataset of images or shapes, for which a user can specify pair-wise similarities, but cannot encode these in terms of high-dimensional attributes [45]. In such cases, we can design f to use only the pair-wise similarities, encoded as a distance matrix $d_{ij} = \delta(\mathbf{x}_i, \mathbf{x}_j)$. DR methods that rely solely on distances are also called *multidimensional scaling* (MDS) methods [278]. Examples of MDS methods are Landmarks MDS [246], Pivot MDS [28], Isomap [247], Glimmer [124], and t-SNE [162]. In general, MDS-like methods are a superset of projections, since one can usually compute a (meaningful) distance matrix given the dimensions $\mathbf{x}^j$. However, this is not always desirable: MDS methods can be expensive, as computing and storing the similarity matrix is quadratic in the observation count N.

Linear *vs* nonlinear: A linear DR method implements a function f which is linear in the positions of observations. The most known (and arguably also simplest to implement) such method is PCA. However, linear methods usually have difficulties in preserving distances and/or neighborhoods when the high-dimensional dataset DS is not a two-dimensional manifold with boundaries embedded in D dimensional space [247]. A classical example hereof is projecting to 2D points densely distributed over the surface of a sphere – this cannot be accurately done using linear projections, as a sphere and a 2D compact region with boundary have different topologies. Nonlinear methods are better in this respect, as they can choose to 'deform' the space in different places and in different ways to achieve a better error minimization (distance and/or neighborhood preservation). One of the earliest non-linear DR methods was proposed by Kruskal [140]. Most current state-of-the-art DR methods are of the nonlinear type [130, 162, 222, 247, 263, 272]. However, accurately estimating distance ratios from the low-dimensional projection is typically not possible using nonlinear methods. In our work next, we will consider nonlinear DR methods since their advantage in better preserving distances or neighborhoods for high-dimensional data outweigh their limitations in not being able to estimate actual distance ratios from the resulting projections.

Local *vs* global: The concept of local *vs* global DR methods is closely related to linearity. Global methods typically define the mapping f as a single transformation in which all coordinates of all observations play a symmetric role. A simple example of such methods is PCA, which is also a linear method. However, global methods have the undesired property of influencing the embedding (placement in low-dimensional space of an observation) by *all* observations in the dataset. As such, even small changes to a subset of observations can influence the overall projection result, which is suboptimal from a stability and robustness perspective, and makes it hard for users to preserve their 'mental map' of the visualization [214]. Moreover, the computation of f becomes in this case expensive, for example, quadratic in the number of observations N [140, 222].

To accelerate the computation of f, and also allow a more local control of the embedding, local methods have been proposed. In brief, such methods select a small subset of so-called landmark or representative observations $ds \subset DS$, and first map these to low-dimensional space, using any existing (and typically high-quality) DR technique. Next, the remaining observations $DS \setminus ds$ are mapped to low-dimensional space by considering only their closest landmark(s) – a process bearing resemblance to data interpolation by local-support basis functions (see *e.g.* [271], Ch. 3). Since $|ds| \ll |DS|$, local methods are very fast, up to $O(N)$ for N observations. Methods in this class include the Least Square Projection (LSP) [202], Local Affine Multidimensional Projection (LAMP) [130], Piecewise Laplacian Projection (PLP) [199], t-SNE [162], Landmarks MDS [246], and force-directed approaches similar to those used to lay out graphs [40, 132, 263]. However fast, local methods require a 'good' selection of landmarks that represents well the structure of the high-dimensional dataset DS. What precisely a good representation is, and how to perform the landmark selection to achieve this, is still an open problem [130]. In our work next, we will consider local methods, since they deliver an overall better embedding (preservation of high-dimensional data structure in the low-dimensional projection) than global methods.

3.3 TRAIL VISUALIZATION

Trail visualization can be briefly described as the process of creating visual representations of datasets consisting of sets of trails T_k, such as defined in Sec. 3.1.2. Interestingly, we can classify trail visualization methods along the same two axes used for multidimensional data visualization, *i.e.*, observation-centric and dimension-centric methods (see Secs. 3.2.2 and 3.2.1, as follows.

3.3.1 *Observation-centric trail-visualization methods*

Observation-centric trail visualizations are simple to understand: Given a trail T_k (as in Eqn. 3.2), one can simply display the sequence of observations $\mathbf{x}_i \in T_k$, suitably annotated by their time attribute x_i^t, so that one can easily infer their temporal order. Probably the simplest way to achieve this is to draw a polyline (or similar curve) $(\mathbf{x}_0, \ldots, \mathbf{x}_T)$, where $\mathbf{x}_i \in T_k$, $1 \leq i \leq T$, and $\forall 1 < i < j < T : x_i^t < x_j^t$. This type of trail visualization is as old as scientific visualization [172]. Early (but still relevant) trail visualizations include stream objects used to depict flow fields in 2D and 3D, such as streamlines, streaklines, and particle paths [228]. While its main appeal is simplicity of interpretation and implementation, this 'static' form of trail visualization can make it hard to show the direction (along the trail) of the tracked objects, as well as their (relative or absolute) speeds. One way to solve the above issues is to code direction and/or speed along the trail into color and/or transparency (see *e.g.* [271], Ch. 5). A more intuitive – and actually pre-attentive – way to show direction and/or speed is to use *animation*: Here, one draws 'trains' of particles that move along all trails T_k in a given dataset. When the trails are suitably densely sampled by particles, and the set $\{T_k\}$ also densely covers the considered spatial domain, the obtained effect is that of a dense moving texture [269, 292]. However, a limitation of animation-based methods is that they create dynamic clutter when too many trails T_k spatially overlap over large extents.

Trail visualization is also a very relevant topic in information visualization. Here, trails are typically describing positions of monitored objects, rather than artificial seeds (like in flow visualization), such as vehicles (*e.g.* planes [112] and ships [225]), see also Fig. 3.6. The infovis aspect of the problem is present in the fact that such trails are annotated by multiple (quantitative but also categorical) attributes, such as *e.g.* a plane's flight-ID, cruising altitude and speed, vehicle type, and actual trajectory spatial location. Several visualization techniques are used to explore such data, as

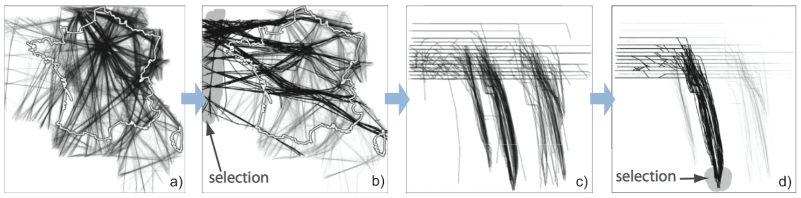


Figure 3.6: Flight trail visualization (a) and interactive exploration (b,c,d). Images taken from [118].

follows.

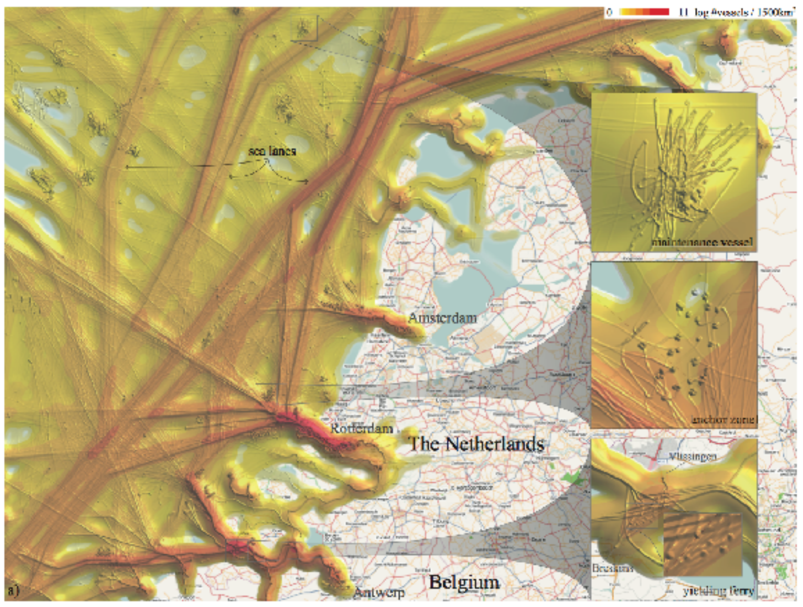


Figure 3.7: Ship trail visualization using shaded density maps. Image taken from [118].

Interaction: The earliest, and still most used, way to address the multivariate visualization problems related to attributed trails is to use interaction, to select and highlight groups of attributes and/or attribute-values of interest in the analysis. Figure 3.6 shows such a visualization constructed by the FromDaDy system [112, 118]. The first image (a) shows a set of about 20000 flights crossing the French territory over the period of one week. The user can select a particular group of flights to examine in more detail (b). For the selected flights, one can change the view to display the flights' latitude data *vs* their cruising height (c). The selection can be next refined (d), followed by the creation of additional views to examine the selected flights.

Density maps: A similar use-case is covered for visualizing ship trajectories by Willems *et al.* [225, 294]. However, a very different solution type is used here: A density map is computed to capture the local spatial density of the moving vessels (Fig. 3.7). The map is suitably colored, pseudo-shaded, and textured to emphasize the density of moving ships as well as their individual trajectories. The key advantage of density maps is their ability to *summarize* very large numbers of trajectory into a single (continuous) image, whose resolution offers a multiscale aspect to the visualization, akin to the early idea introduced by De Leeuw and Van Liere for the compact visualization of graph drawings [282]. Such techniques share also a commonality with the earlier-mentioned dense flow-visualization techniques [269, 292] in the sense that the resulting view consists of a single continuous image, rather than a (large) group of individual trails. As such, these techniques are also known under the name *image-based* techniques. We will also use density maps to similar ends in our image-based techniques proposed in Chapters 7 and 8.

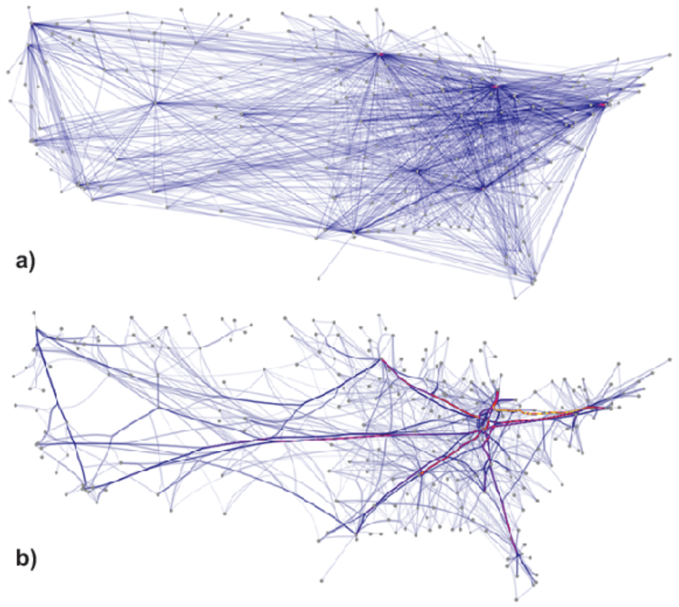


Figure 3.8: US airlines graph (a) and its bundled drawing (b). Images taken from [108].

Bundling techniques: However effective in summarizing local density for a large set of trails, image-based techniques typically remove the original data items (trails) from the final visualization. This is not desirable when one wants to reason about specific trails rather than about their aggregated density [118]. A different way to summarize large sets of trails is offered by *bundling* techniques. These have been originally introduced in the context of summarizing large node-link graph drawings [107]: To simplify such drawings, and also reduce the edge-crossing clutter, edges which run largely parallel to each other and at close distance are bent to be grouped in a so-called bundle, akin to cables being joined in electrical installations. A key observation is that bundling is not restricted to graphs (or, more precisely, to graph *drawings*), even though this may appear so from the terminology used in the literature. Indeed, bundling can be applied to any drawing consisting of elongated objects, such as trails, which are amenable to summarization by the aforementioned bending and grouping. In the last decade, many such graph-drawing and/or trail bundling techniques have been proposed for hierarchical compound graphs [107, 267] and general graphs and trail-sets [53, 76, 92, 108, 142, 143]. Figure 3.8 shows a graph indicating the connections between airports (nodes) by airlines (edges) over the US territory, and its bundling result. As visible, the strong connection patterns between (groups of close) airports are now easier to discern.

Trail bundling has been recently extended to several additional applications. Several methods treat dynamic data, *i.e.*, trail-sets whose content changes in time. This allows one to visualize very large datasets whose elements (trails) can be chronologically ordered, using either animation techniques or small multiples showing several 'keyframes' from such animations [117, 187]. Interactive exploration techniques are proposed to morph between the bundled and unbundled (raw) trails, so as to reduce the inherent distortion caused by bundling and show the actual location of the trails [112, 139]. Particle animation along the (un)bundled trails can be used to indicate the direction of tracked shapes [118]. Directional bundling can be used to separate trails running in opposite directions from being bundled together, which allows next one to color-code bundled to show their direction [206, 231]. Pseudo-shading, similar to the one used for density maps [294], can be used to emphasize bundles better and help one visually follow them end-to-end over crossings [76, 267]. Finally, bundling has also been used to produce simplified views of eye-tracking data, where the trails indicate how a subject's gaze moves when scanning an image [7, 119, 206]. For a recent overview of the state-of-the-art in graph and trail bundling, we refer to Lhuillier *et al.* [152]. Trail bundling, including most visualization options described above, will be the topic of our novel visualization proposals in Chapters 8 and 7.

Separately, we note that trail visualizations can be also classified in physical *vs* abstract-space. Physical trail visualizations have as target trajectories created by objects moving in physical (2D or 3D) space[1]. All trail visualizations mentioned so far are in this category. Abstract-space trail visualizations have as target entities that move in abstract, possibly high-dimensional, spaces. Examples of such trail visualizations are those of change of software artifacts [245] or of activations of neurons in artificial

1 We refer here to 'physical' space in the same sense as it is referred when describing scientific visualization as opposed to information visualization.

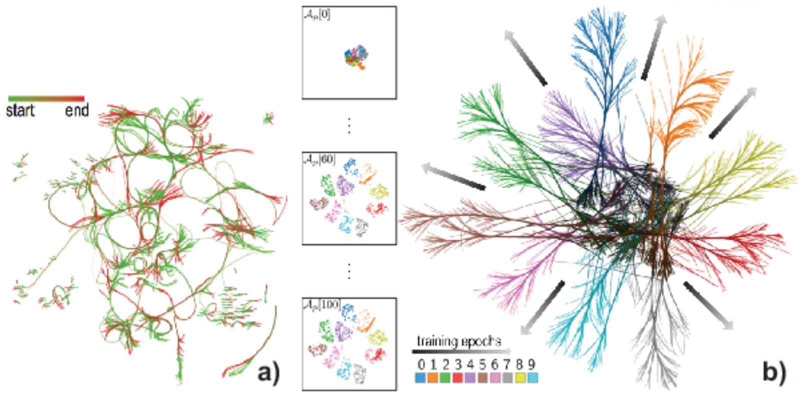


Figure 3.9: Bundled visualizations of abstract trails. (a) Evolution of software entities attributed by their quality metrics (image from [245]). (b) Evolution of neurons during the 100-epoch training of an artificial neural network (image from [212]).

deep neural networks [212] – see Figs. 3.9a,b respectively. In both these cases, the tracked objects also represent abstract, rather than physical, items. These items are described by a high number of attributes, and the visualization aims at showing how they change in time. One salient aspect of such abstract spaces and their trails is that direction and absolute position in the visualization does not have a particular meaning, as opposed to direction and absolute position in 2D or 3D physical space. Given the high dimensionality, trails for these objects can be drawn using multidimensional projections, such as those described in Sec. 3.2.2. We will use such techniques in Chapter 6 for a novel use-case – the exploration of the high-dimensional parameter space of a shape-tracking system.

3.3.2 *Dimension-centric trail-visualization methods*

In contrast to observation-centric methods for trail visualization (Sec. 3.3.1), dimension-centric methods focus on showing how the individual attributes x_i^j of points $\mathbf{x}_i$ in one or more trails T_k (Eqn. 3.2) change in time. Dimension-centric methods are relatively more effective for visualizing trails of abstract, rather than physical, objects. Indeed, abstract objects do not have dimensions representing spatial position, so observation-centric methods such as projections (Sec. 3.3.1) can be quite challenging to use, since the depicted trajectories take place in a highly abstract space. In such spaces, only inter-observation distance has an intuitive meaning, as compared to observations in a physical space, where the (typically 2D or 3D) Cartesian coordinates of the visualization map one-to-one to physical dimensions (attributes) of the trails.

Globally speaking, such dimension-centric methods are very related to the dimension-centric visualization methods for multidimensional data discussed in Sec. 3.2.1. The table lens method illustrated in Fig. 3.1b is precisely an instance of such a method. The rows of the table indicate recordings of the prices of several given stocks over a period of about a month, with a minute-level resolution [264]. The leftmost column (1 in Fig. 3.1) indicates the recording time. Columns 4 to 7 in the same figure indicate various stock attributes. If the table lens is sorted by this column, as shown in the figure, then the display of columns 4 to 7 are basically four graphs of the time-dependence of these attributes.

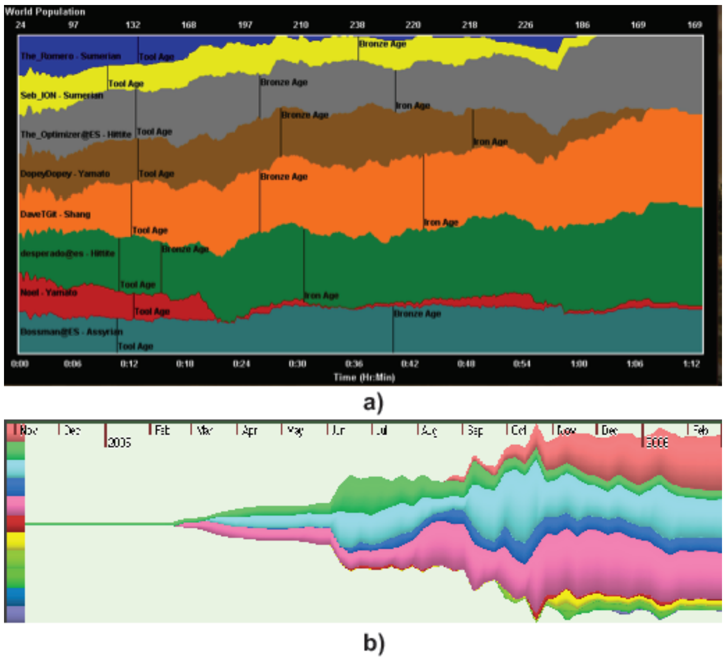


Figure 3.10: Stacked area charts for trail visualization. (a) Evolution of the strength of several players in a multiplayer game [180]. (b) Evolution of the coding effort of a software development team (image created with the SolidTA tool [218, 287].

Stacked area charts are a related visualization method [185]. The spatial encoding of these charts is identical to the Cartesian one proposed by the table lens: One dimension of the 2D layout represents time, while the 'thickness' of the bars, mapped to the other dimension, represents data values. The key difference between table lenses and stacked area charts is that the latter does not leave any empty space between the (colored) bars representing the values of all attributes at a given time moment (therefore the 'stacked' concept), whereas table lenses align all values x_i^j of a dimension j along the time axis.

Figure 3.10a shows a stacked area chart used in Microsoft's (once) popular game Age of Empires [180]. Here, the horizontal axis represents time, and each colored 'timeline' strip represents the total population of one of the game players over the play time. Stacking allows one to easily see when the game has been most intense (sum of all players' populations is maximal); and also removes empty space between the timelines, thereby making it easier to spot dominating and weak players at every moment and over time. Figure 3.10b shows a rather more scientific usage of stacked area charts (though, one which was inspired from the aforementioned game). Here, the horizontal axis represents time, and the vertical axis shows the relative and total development effort of programmers contributing to a software repository over the repository's lifetime [218, 287]. Here, the total thickness of the chart (at a time moment) indicates development effort of the entire team; and the relative thicknesses of each developer timeline allow one to compare development effort across one or several programmers at the same or different time moments.

Essentially, both examples in Fig. 3.10 can be seen as trails, over time, of the *state* of a system, described by its various attributes (*e.g.* players' strengths or developers' contributions). Another well-known example of using stacked area charts to depict evolution of a system's state is the ThemeRiver tool that visualizes how topics of interest change over time for a large document collection [104]. In Chapter 5, we will use dimension-based trail visualizations similar to table lenses and area charts to examine the evolution in time of the state of a 3D shape tracker.

4 IMAGE SEGMENTATION BY DENSE SKELETONS

As extensively described in Chapter 2, before (even) being able to perform tracking of shapes from a 2D image or video stream, a critical step is separating such shapes – or, more formally put, their projections or views in a two-dimensional image, froun the surrounding background. To this end, numerous methods have been designed, see *e.g.* [46, 128, 175, 211, 236, 277]. However, most such methods have various limitations in the sense of the types of shapes they can segment from the existing background; accuracy of segmentation (in terms of robustly identifying relevant parts of the shapes from the surrounding background); ease of use (in terms of amount of input asked from the user before the segmentation method is applicable); and efficiency, measured in terms of computational scalability *vs* resolution of processed images, for instance[1].

Why is this problem relevant you our context outlined in Chapter 1? We find several reasons to this end, as follows. First and foremost, our thesis scope is the analysis of dynamics of moving shapes. Clearly, to be able to say anything about such shapes, we need to be able to *quantify* them. This means, first and foremost, separating them from surrounding details in data-acquisition datasets, such as video streams. Hence, a good (robust) segmentation method is critical. Secondly, our concrete use-case for *RQ*1 (Sec. 1.3) is strongly constrained by the *topology* of the shapes of interest. Revising the description of our use-case outlined in Sec. 1.3, we are looking to track very specific shapes – teats of a cow, which, although variable, have a quite specific geometric and topologic signature.

As such, the detection of specific types of shapes in general-purpose 2D images is clearly within our thesis' research scope. In this chapter, we present our work towards the segmentation of shapes from 2D luminance images, which uses structural (topological) properties of the shapes. Besides being innovative in terms of the used techniques, we argue that our proposed segmentaton method can be a first step towards the effective tracking of complex/articulated natural shapes from 2D video sequences.

4.1 INTRODUCTION

Skeletons, or medial axes, are well-known 2D shape descriptors used in many applications in shape analysis and classification, shape recognition, shape matching, topological analysis, image registration, and path plan-

1 The material in this chapter is based on the following publication: M. van der Zwan, Y. Meiburg, and A. Telea. A dense medial descriptor for image analysis. In J. Braz, S. Battiato, and F. Imai, editors, *Proc.* 8th *IEEE International Conference on Computer Vision Theory and Applications (VISIGRAPP)*, volume 1, pages 133–140, 2013.

ning. Medial axis structures, augmented with distance information from the medial axis to its corresponding shape, generate the so-called Medial Axis Transform (MAT), which is a true dual for the input shape. In other words, the MAT can be used for the exact reconstruction and also for the simplification of shapes at user-specified levels of detail. 2D skeletons and MATs have been extended to three dimensions to create surface and curve skeletons and their corresponding medial surface transform (MST), which allow processing of 3D shapes analogously to their 2D counterparts. In this chapter, we focus on 2D skeletons and MATs.

However powerful, skeletons and MATs have the crucial limitation that they require as input a digital shape, *i.e.* a closed boundary which divides the embedding space into inside and outside regions. This limits their direct application to datasets containing pre-segmented shapes. However, in many applications, one has continuous fields as inputs, such as grayscale or color 2D images or 3D scalar volumes such as CT or MRI scans. Although pre-segmenting such datasets into binary shapes and further using skeletons to analyze such shapes is possible, this is a non-trivial process which requires *a priori* knowledge on the nature and position of the shapes of interest. Moreover, since skeletons require binary shapes, they cannot directly handle fuzzy shapes whose boundaries are defined by a range of scalar values. Eliminating this limitation, *i.e.* enabling skeletal and MST descriptors to directly handle grayscale images, can open new ways for using the analytic power of such descriptors for image segmentation, editing, and classification applications.

In this chapter, we present a framework for representing and manipulating 2D images using a new descriptor: Dense medial axes. Our framework operates in three steps. First, we decompose a grayscale image into several so-called threshold sets T_i, *i.e.* pixels whose values exceed a given set of scalar values v_i. Next, we compute a simplified medial axis transform M_i of each threshold set T_i, using a suitable simplification value τ_i. Finally, we use the medial transforms M_i to perform several types of image processing operations on the initial image, ranging from perfect image reconstruction to image simplification, segmentation, editing, and artistic painting effects. The set of threshold-values v_i and medial simplification values τ_i effectively create a two-dimensional scale-space in which we encode the image luminance variations and shapes present in the image, respectively. We propose an efficient GPU-based implementation of our dense medial descriptors, which can compute these and the associated image processing operations in real-time on mega-pixel images. We demonstrate our framework with several image processing applications.

The structure of this chapter is as follows. Section 4.2 overviews related work on 2D medial descriptors. Section 4.3 details the three steps of our framework: threshold-set computation (Sec. 4.3.1), medial transform computation (Sec. 4.3.2), and image reconstruction (Sec. 4.3.3). Section 4.4 presents ways in which we can parameterize the above-described steps of our pipeline to achieve several types of image processing operations, and illustrates these with examples. Section 4.5 discusses our framework. Section 4.6 concludes the chapter.

4.2 RELATED WORK

Given a two-dimensional binary shape $\Omega \subset \mathbb{R}^2$ with boundary $\partial\Omega$, we first define its *distance transform* $DT_{\partial\Omega} : \Omega \to \mathbf{R}_+$ as

$$DT_{\partial\Omega}(\mathbf{x} \in \Omega) = \min_{\mathbf{y} \in \partial\Omega} \|\mathbf{x} - \mathbf{y}\| \tag{4.1}$$

The skeleton, or medial axis, of Ω is next defined as

$$\begin{aligned} S(\partial\Omega) = \quad & \{\boldsymbol{x} \in \Omega | \exists \boldsymbol{f}_1, \boldsymbol{f}_2 \in \partial\Omega, \boldsymbol{f}_1 \neq \boldsymbol{f}_2, \\ & \|\boldsymbol{x} - \boldsymbol{f}_1\| = \|\boldsymbol{x} - \boldsymbol{f}_2\| = DT_{\partial\Omega}(\boldsymbol{x})\}, \end{aligned} \tag{4.2}$$

where $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ are the contact points with $\partial\Omega$ of the maximally inscribed disc in Ω centered at $\boldsymbol{x}$, also called *feature transform* (FT) points [255] or *spoke vectors* [253], where the feature transform is defined as

$$FT_{\partial\Omega}(\boldsymbol{x} \in \Omega) = \arg\min_{\mathbf{y} \in \partial\Omega} \|\mathbf{x} - \mathbf{y}\|. \tag{4.3}$$

The skeleton, together with the distance transform, form the Medial Axis Transform (MAT), which can be used to exactly reconstruct the input shape Ω [240, 268].

Two-dimensional MAT techniques can be classified into three groups. *Geometric* methods use a polygonized version of $\partial\Omega$ to compute its Voronoi diagram and the skeleton as a subset thereof [189]. *Thinning* methods iteratively remove $\partial\Omega$ pixels while preserving connectivity [194]. Pixel removal in distance-to-boundary order enforces centeredness [208]. Such methods are simpler than geometric methods and they also directly use a pixel-based image representation. *Distance field* methods find the MAT along singularities of $DT_{\partial\Omega}$ [106, 223, 241, 268, 288], and can be efficiently implemented on GPUs [33, 65, 255, 256]. General-field methods use fields smoother (with less singularities) than distance transforms [3, 48, 103], thus are more robust for noisy shapes. Foskey *et al.* compute the θ-SMA, an approximate simplified medial axis, using the angle between feature vectors [87]. The θ-SMA can get disconnected along the so-called ligature branches. An accuracy comparison of different field methods for 2D distance and feature transforms is given in [216].

Clear, or regularized, skeletons are extracted from noisy shapes by thresholding *importance measures* to prune skeleton pixels caused by small shape details [235]. One of the simplest, and most effective, such measures is the collapsed boundary length metric $\rho : S \to \mathbb{R}_+$, which ranks skeleton pixels $\mathbf{x}$ by the boundary length, along $\partial\Omega$, between their feature points [52, 189, 268], *i.e.*

$$\rho(\mathbf{x} \in S) = \|\partial\Omega(\mathbf{f}_1, \mathbf{f}_2)\|, \tag{4.4}$$

where $\mathbf{f}_1$ and $\mathbf{f}_2$ are the feature points of the skeleton point $\mathbf{x}$, and $\|\cdot\|$ denotes the arc-length shortest distance along $\partial\Omega$ between these points (for feature points belonging to disconnected boundary fragments, this distance is set to infinity). The metric ρ increases monotonically on skeletons of genus 0 shapes from their periphery to their center, so thresholding it is guaranteed to directly yield a connected skeleton [52, 268].

However effective for shape registration [257], matching [15, 69], classification [52], and recognition [165], skeletons can only be computed for *binary* shapes Ω. Given a grayscale image, Ω can be computed using various segmentation methods [46, 133, 154, 236]. Simpler, but more generic, segmentation methods include classical level-sets and threshold-sets [234], which are nested structures that capture all image points whose grayvalue is equal to, or respectively larger than, a given value. However, segmentation poses two problems. First, we need to know, in advance, which shapes we are searching for in a given image. Secondly, this approach only delivers skeletons of the image subset captured by the segmentation. In other words, we do not have a medial descriptor for the *entire* image, *e.g.*, we cannot talk about the skeleton of a fuzzy shape or a grayscale image.

4.3 PROPOSED FRAMEWORK

We propose to join the grayvalue information present in threshold set descriptors with the shape information delivered by medial descriptors in a single new *dense* medial descriptor (DMD), as follows (see also Fig. 4.1). First, we reduce an image to a set of threshold-sets (Sec. 4.3.1). Secondly, we compute a simplified MAT for each threshold-set (Sec. 4.3.2). Finally, we use the threshold-set grayvalues and computed MATs to generate our DMD (Sec. 4.3.3), which we can use next for various image processing operations (Sec. 4.4).

4.3.1 *Threshold set computation*

Given a grayscale image $I : \mathbb{R}^2 \to \mathbb{R}_+$ and a grayvalue $v \in \mathbb{R}_+$, we define the threshold-set

$$T(v) = \{\mathbf{x} \in \mathbb{R}^2 | I(\mathbf{x}) \geq v\}. \tag{4.5}$$

By definition, threshold-sets for increasing grayscale values are nested 2D structures. For a digital n-bit-per-pixel image, we have thus 2^n threshold sets, or layers, $T_i = T(i), 0 \leq i < 2^n$. Here and further, we use a value of $n = 8$. Further, from each layer, we remove foreground and background islands with an area ϵ smaller than 3% of the layer's area $|T_i|$. This further simplifies our medial descriptors used to encode the layers (Sec. 4.3.2).

The density of layer borders is proportional with the probability of having an edge in the image: Low-density areas indicate that consecutive layers are far apart, thus the image is relatively flat. High-density areas indicate close consecutive layers, *i.e.* quickly varying grayvalues. We will further

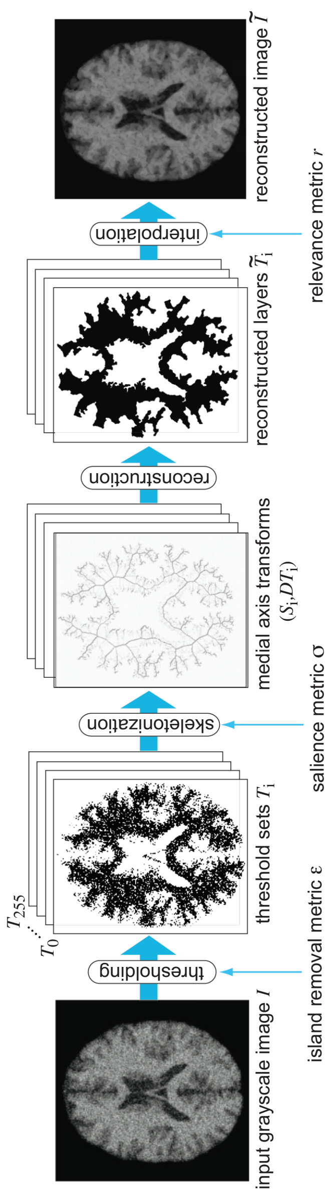


Figure 4.1: Dense medial descriptor computation pipeline.

use this observation in the reconstruction pass (Sec. 4.3.3). Threshold-sets are nested, *i.e.* $\forall i < j, T_j \subset T_i$. This observation is important, as it implies that if we remove a pixel from T_j, this pixel will get the (darker) grayvalue of its closest parent layer.

4.3.2 *Simplified medial axis*

A threshold-set T_i can be seen as an arbitrary 'slice' in the image grayvalue space. Geometrically speaking, T_i can have any shape, *e.g.* a collection of noisy disconnected components. We propose here to use MATs to capture the essence of the shape of a T_i and remove its spurious details.

For this, we compute the distance transform $DT_i = DT(T_i)$ and simplified skeleton $S_i = S(T_i)$, following Eqns. 4.1 and 4.3 respectively. We compute DT_i using the GPU-based exact Euclidean method of Cao *et al.* [33]. This method also computes the feature transform of a shape (Eqn. 4.3). Hence, it is trivial to modify this method to determine, for each point $\mathbf{x} \in \mathbf{T_i}$, which are its two feature points, and next, following a simple arc-length parameterization of ∂T_i, analogous to [268], the collapsed boundary length at $\mathbf{x}$.

As noted earlier, T_i can contain a large amount of geometrical and topological noise. Once we have S_i and DT_i, we can remove these easily. As skeleton importance, we use the so-called *salience metric*

$$\sigma : S \to \mathbb{R}_+ = \rho / DT, \tag{4.6}$$

equal to the collapsed boundary length ρ (Eqn. 4.4) divided by the distance transform (Eqn. 4.1) [266]. This metric has the desirable property that it removes small-scale boundary noise, but it keeps salient features, such as cusps or dents. Figure 4.2 illustrates this: The input image is an 8-bit noisy human brain CT (a), from which we select the threshold-set corresponding to the level 132 (b). Next, we remove small foreground and background islands (as mentioned in Sec. 4.3.1), compute the simplified MAT of this threshold-set, regularized by the saliency metric σ, and reconstruct this set from this skeleton. The result (c) captures the main shape described by the threshold-set (b), ignoring small-scale details such as specks, holes, and boundary noise.

4.3.3 *Image reconstruction*

So far, we have reduced our input grayvalue image I to a set of threshold-sets T_i, each having an MAT S_i. We can reconstruct a simplified version $\tilde{T}_i$ of each T_i (in increasing order of i, *i.e.* from dark to light layers) from its corresponding MAT (S_i, DT_i) by either executing a Fast-Marching-Method (FMM) 'inflation' of S_i outwards until each point $\mathbf{x} \in \mathbf{S_i}$ reaches the distance $DT_i(\mathbf{x})$ [234, 268], or alternatively drawing discs centered at $\mathbf{x}$ with radius $DT_i(\mathbf{x})$, and coloring each layer T_i with the grayvalue i. The first approach (FMM) works best on a CPU, while the second scales better on the GPU. However, this method is basically a nearest-neighbor (zero-order interpolation) reconstruction of I, so it shows intensity-banding effects around the

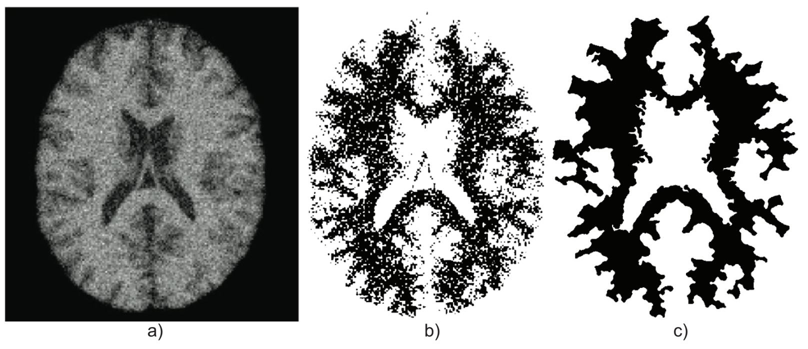


Figure 4.2: Salience metric for skeleton simplification: (a) grayvalue image; (b) threshold set; (c) threshold set reconstructed from saliency skeleton.

boundaries ∂T_i of the layers T_i. A better way is as follows: For each two consecutive layers i and $i+1$, whrere $i \in [0, 255]$, we reconstruct the layers T_i and T_{i+1} as above (either on the CPU or GPU), and next, we set the gray-value $v(\mathbf{x})$ of each pixel $\mathbf{x}$ located between the boundaries ∂T_i and ∂T_{i+1} to

$$v(\mathbf{x}) = \frac{1}{2}\left[\min\left(\frac{DT_{T_i}}{DT_{T_{i+1}}}, 1\right) v_i + \max\left(1 - \frac{DT_{T_{i+1}}}{DT_{T_i}}, 0\right) v_{i+1}\right] \tag{4.7}$$

This achieves a smooth distance-based interpolation between the boundaries ∂T_i (with grayvalue v_i) and ∂T_{i+1} (with grayvalue v_{i+1}) in the Hausdorff sense. Applying Eqn. 4.7 at all pixels $\mathbf{x}$ of the image space yields our final reconstructed image $\tilde{I}$. Examples of reconstruction are discussed next in Sec. 4.4.1.

4.4 APPLICATIONS

We next present several applications of our dense medial descriptor.

4.4.1 *Reconstruction*

Overall, the interpretation of our reconstruction technique is simple: Given several layers T_i with corresponding MATs (S_i, DT_i), we can reconstruct a smooth version of the original image I from which T_i have been produced. Thus, I is integrally encoded in our dense MAT (S_i, DT_i). For instance, if we encode *all* layers T_i present in the original image, and do not simplify at all the resulting MATs S_i, the reconstruction presented in Sec. 4.3.3 is an *exact* copy of the original image, by definition, *i.e.* since we encode all luminance layers and since an unsimplified skeleton exactly preserves the shape of each layer. Thus, our dense medial descriptor (DMD) can encode the full input information, if desired. However, our DMD can be used to simplify the input image I in several ways, as follows.

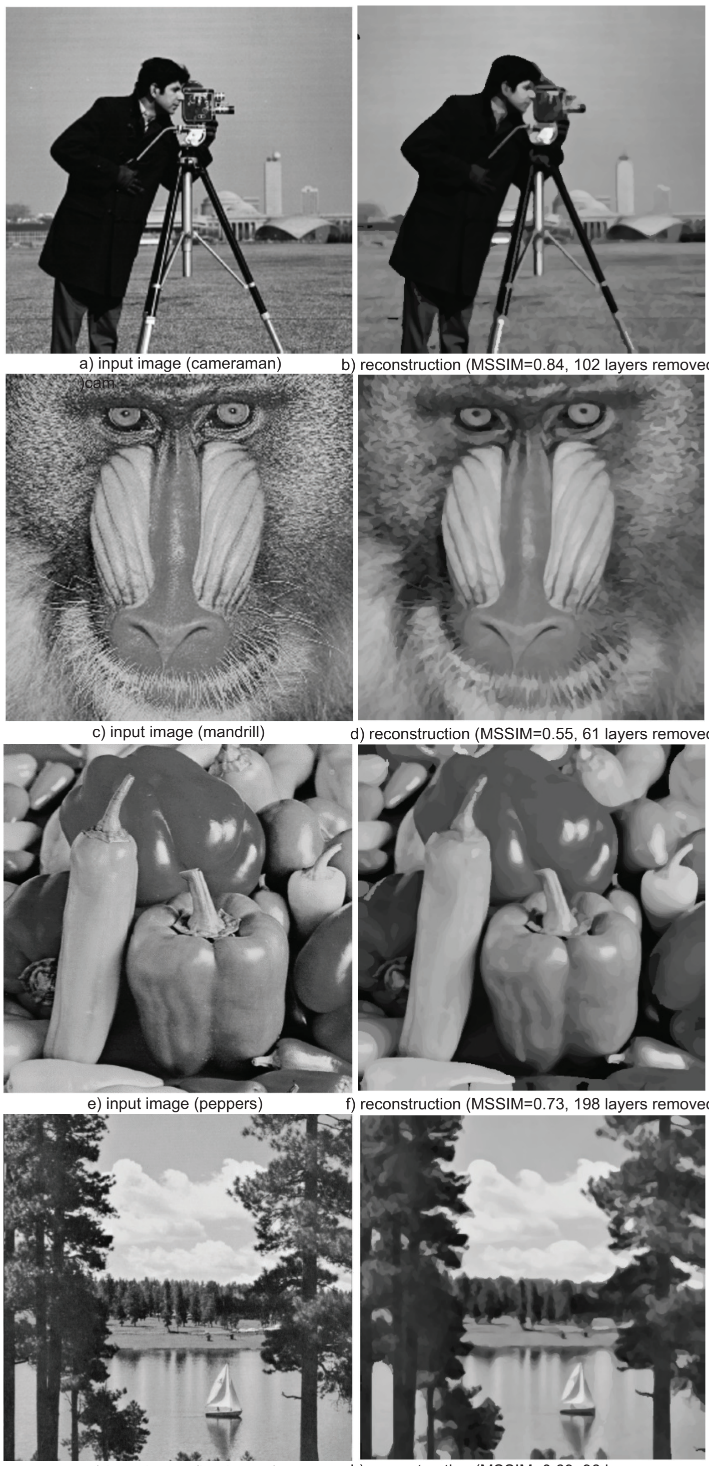

a) input image (cameraman)
b) reconstruction (MSSIM=0.84, 102 layers removed)
c) input image (mandrill)
d) reconstruction (MSSIM=0.55, 61 layers removed)
e) input image (peppers)
f) reconstruction (MSSIM=0.73, 198 layers removed)
g) input image (landscape)
h) reconstruction (MSSIM=0.69, 96 layers removed)

Figure 4.3: Image reconstruction accuracy (*vs* SSIM) while removing layers.

First, we can only encode the *relevant* layers T_i. A layer is deemed relevant if its removal from the reconstruction (Sec. 4.3.3) causes a too large difference between the original image I and the reconstructed image (Eqn. 4.7). Indeed, the advantage of our reconstruction scheme is that it allows to easily remove, or keep, layers T_i in the reconstruction process. As such, given a typical image with 255 layers, we can decide on-the-fly which layers are relevant for the reconstruction or not, depending on application-specific metrics.

The simplest of such metrics is the *relevance* of a layer: Given all layers T_i, we can remove those which contribute less to reconstructing an image close to the input image I. To compare the reconstruction $\tilde{I}$ with the original image I, we use the well-known mean structural similarity index (SSIM) metric [289]. Figure 4.3 ilustrates this. Here, we have removed the least relevant layers to the reconstructed image (as according to SSIM) and plotted the SSIM metric. We see that we can remove around 30..50% of the 255 layers of an 8-bit image without a perceptual decrease in image quality: Details such as salient sharp boundaries, highlights, and even global small-scale patterns (such as the mandrill's hair structure) are well preserved. Accordingly, this means we can *compress* an image, by the same layer removal ratio, *i.e.* 60% (a), 23% (b), 78% (c), and 37% (d) respectively, with little *perceptual* loss. Consequently, if we accept the implied perceptual difference, between the input image and our simplification, techniques such as JPEG encoding can be subsequently applied atop of our simplification.

4.4.2 *Segmentation*

Segmenting an image into its salient shapes has countless applications in medical imaging, computer vision, and image classification. We show below how our DMD representation can be used for image segmentation. Given an image I with $0 \leq i < 256$ grayvalues, we compute, for each layer T_i, its relevance r as being the difference between I and the reconstruction using all layers except T_i. Next, we select for reconstruction only the k most relevant layers, where k is a small user-supplied value. This will keep the most 'salient' shapes present in the image. Moreover, since the shape of each layer is simplified by island removal (Sc. 4.3.1) and boundary jaggies removal (Sec. 4.3.2), the resulting reconstruction will have simpler shapes.

Figure 4.4 shows an example. Statistics of the input image (a) are displayed in Fig. 4.4e. The area $|T_i|$ of a layer is computed as the pixels that have precisely grayvalue i. We notice, as expected, that the darkest 20% layers are empty and brighter layers have increasingly less pixels (highlights are smaller than darker zones). The number of shapes (connected components) per layer is relatively large for the layers having a large area, which indicates that the image is non-trivial to segment. The relevance r shows several local peaks: These are layers which are (1) globally significant for the image representation and (2) more significant than layers having similar grayvalues. We select the $k = 6$ most significant such layers as shown in Fig-

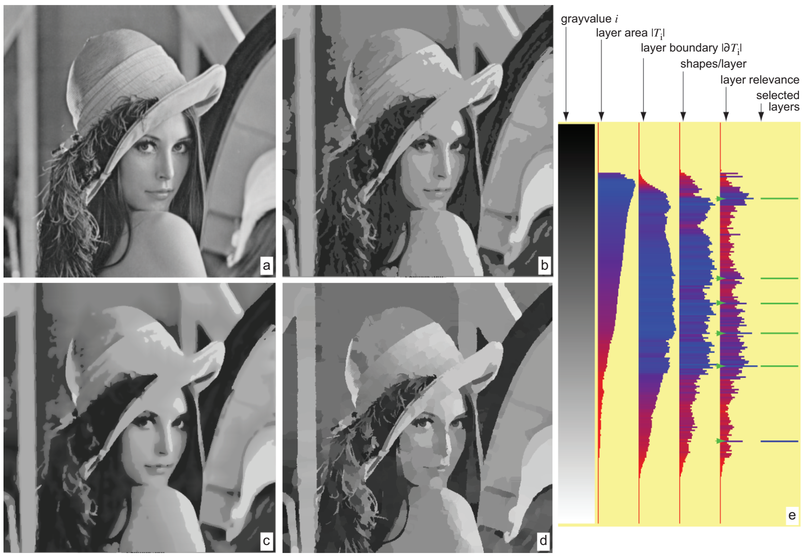


Figure 4.4: Segmentation example. (a) input image; (b) 5 most relevant layers selected; (c) reconstruction; (d) mean shift segmentation comparison; (e) layer statistics visualization (see Sec. 4.4.2).

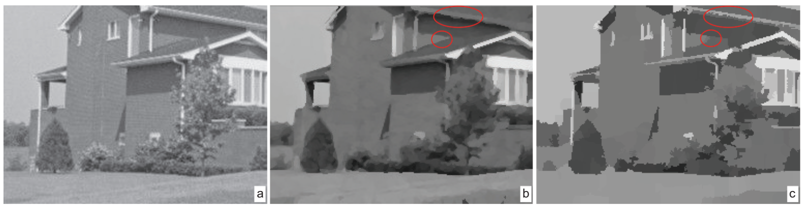


Figure 4.5: Segmentation example. (a) input image; (b) our method (60% layers); (c) mean shift segmentation (see Sec. 4.4.2).

ure 4.4b. Here, we do not use the linear interpolation of consecutive layer grayvalues (Eqn. 4.7), so the result shows a luminance-quantization-like segmentation of the image. In contrast, using linear interpolation (Fig. 4.4c) blurs the reconstruction where the original image has low contrast (since, as explained in Sec. 4.3.1, layers in such zones have far-apart boundaries) but keeps sharp luminance edges visible (since these correspond to high-density layer boundaries). This yields a fuzzy segmentation of the input image. For comparison purposes, Fig. 4.4d shows the result of mean-shift segmentation [46] applied on the input image. Although the produced segments are not identical to ours (Fig. 4.4b), the overall segmentation impression is similar.

The exact selection of the k most important layers is not critical for the segmentation results. Figure 4.5 shows an example. Here, we reconstructed the input image (a) by using the 60% most relevant layers. The result (b) is quite similar with the results of mean shift segmentation (c), see *e.g.* the segments corresponding to the house walls, window panes, and bush. Our segments are less jagged than the ones produced by mean shift *and* still preserve their salient sharp corners, see *e.g.* the areas marked in red on the figure. The reason for this is the working of the salience regularization metric for medial axes, which, as explained, eliminates small boundary jaggies but keeps salient corners unchanged. On the other hand, our result (b) has a slightly more fuzzy segmentation aspect than mean shift (c). If a more clear-cut segmentation is desired (less segments), fewer most-relevant layers can be selected, as shown in the earlier example (Fig. 4.4).

Figure 4.6 shows a final segmentation example. The input image (a) shows a skin lesion (naevus) photograph taken with a Handyscope mobile dermatology device in the framework of a digital dermatology skin-cancer screening project. A typical network naevus structure is visible herein. The central part of the naevus has a slightly darker, and denser, network pattern, which is only visible on the original high-resolution 1936 by 2592 pixels image. The marked boundary (in green) shows the segmentation of the lesion as manually drawn by a dermatology expert atop of this image. We processed the input image, without the manually-drawn segmentation, to obtain the result in Figure (b). Here, we see the three most relevant layers segmented from the input image, *i.e.*, the lesion's extent atop of the healthy skin (A), and two regions corresponding to the darker and denser central area (B,C). This figure was obtained with relatively low salience and island-removal values, $\epsilon = 0.03$ and $\sigma = 2$ (Secs. 4.3.1 and 4.3.2). If we increase these values to $\epsilon = 0.05$ and $\sigma = 5$, more small-scale islands and also jaggies on the layers' boundaries get removed. Figure (c) shows the result: The lesion's outer boundary has now been considerably smoothed. Note, also, that this layer is indeed by far the most relevant from all the image's layers, as indicated by its large relevance value (Fig. 4.6d (A)), and its shape is quite similar to the manual segmentation. The lesion's inner layers are also simplified, but to a lesser extent.

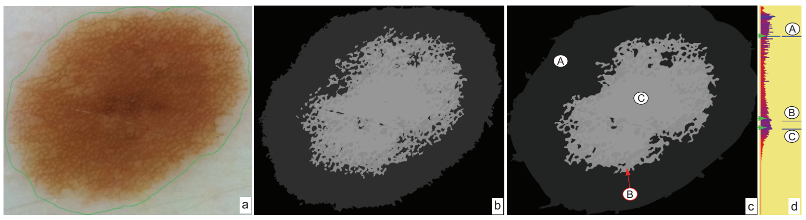


Figure 4.6: Skin image segmentation. (a) original image showing manual segmentation; (b) detail-preserving automatic segmentation; (c) simplified automatic segmentation with corresponding relevance metric (d).

The target users (dermatology medical specialists) noted that the tool can be very useful as a guided aid to their manual work rather than an automatic segmentation techinque: The relevance values suggest salient structures in the input images. Seeing such values, users select them in the relevance metric-bar, visualize the corresponding structures, and decide whether these are useful segments of the case under analysis.

4.4.3 *Artistic editing*

Our method can also be used to generate painting-like effects from a given (sharp) photograph, similar to the artistic edge and corner preserving smoothing effect of Papari *et al.* [195]. By increasing the skeleton saliency metric σ (Eqn. 4.6), we eliminate small-scale jaggies of *all* threshold-sets, *i.e.* isophote contours, while keeping their sharp corners. The reconstruction (Sec. 4.3.3) interpolates between these simplified contours, yielding effects akin to painting. Figure 4.7 (c,f) illustrates this on two complex, fine-grained detail, images. The resulting images, where MSTs have been simplified by a saliency value of $\sigma = 0.4$ and we kept 65% of the original threshold-sets, show a painting-like effect of the input forest images, where small-scale details are 'clustered' into larger shapes (due to the skeleton simplification), but the contrast is not unnecessarily blurred (due to keeping a significant number of the original grayvalues, or threshold-sets). As such, salient details such as the dark thin trees and light spots are well preserved, but small-scale *and* weak-contrast details such as the foliage, are simplified. The painting effect is strikingly similar with the results produced by the method of Papari *et al.*, see Fig. 4.7 (b,e).

4.5 DISCUSSION

Below we discuss several aspects of our method.

Robustness: We use medial axes for saliency-based simplification and encoding of image layers. Although medial axes are known to be unstable and not robust to noise, we should stress that this does not affect our method. In-

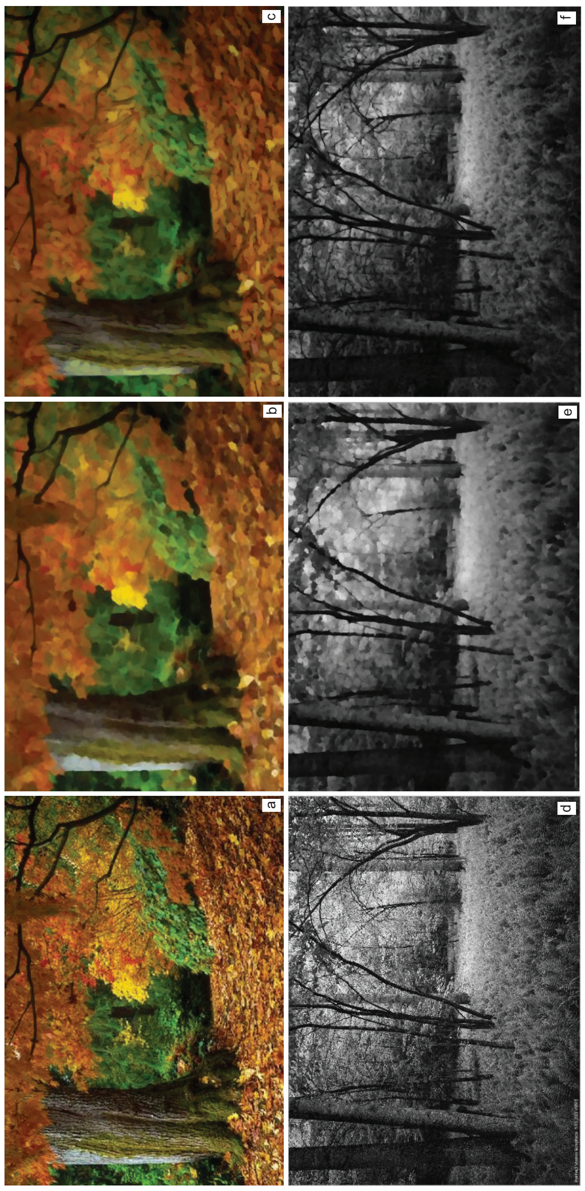


Figure 4.7: Original images: color example (a) and grayscale example (d). Painting-like effects obtained using the method of [195] (b,d) and our method (c,f).

deed, we use *regularized* medial axes, *i.e.* eliminate noisy branches by means of the salience metric (Eqn. 4.6). As explained in detail in [266, 268], this regularization produces medial axes which are robust to arbitrary boundary noise for shapes of arbitrary genus. Also, we should note that the medial axes are exact, *i.e.* precisely centered in their shapes and pixel-thin, by construction, given the exact Euclidean DT we use [33] and the underlying skeletonization algorithm [268].

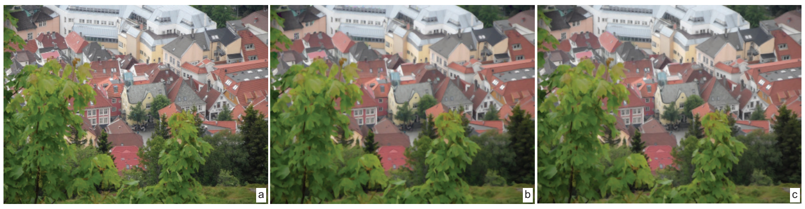


Figure 4.8: Original color image (a). Simplified representation using our method in the RGB space (b) and HSV space (c).

Speed: Our method relies on the fast computation of distance transforms and skeletons (Secs. 4.3.2,4.3.3). On the CPU, we have used for this the method presented in [268], which is worst-case $O(n \log n)$ for an image of n pixels. On the GPU, using the method in [33], we achieve a complexity of $O(n)$. For images of 512^2 pixels, our CPU method takes about 1 minute on a PC at 2.5 GHz, while on an Nvidia 330 GTM GPU, we take 1..2 seconds. The memory complexity is $O(n)$, as we only need to store a fixed set of 256 MSTs per image.

Parameters: Our method selects a subset of relevant threshold-sets from the 256 possible sets, and then computes simplified MSTs for each such threshold-set according to the specified saliency. Hence, saliency and relevance (σ, r) create a two-dimensional scale-space for the input image. Selecting less threshold-sets (high r) emphasizes fewer high-relevance structures in the image (Sec. 4.3.3. Simplifying each MST (high σ) reduces the border-detail of such contours (Sec. 4.3.2). The third and final parameter is the size ϵ of the foreground and background islands to be removed (Sec. 4.3.1). For typical applications, setting ϵ to values between 3 and 5% of the area of a layer achieves the desired effect, *i.e.* removal of small isolated bright or dark specks.

Color images: Applying our DMD representation to color images is trivial. For this, we apply the entire pipeline (threshold sets, medial axes, and simplified reconstruction) to each channel of a color image. Figure 4.8 illustrates this. As visible, choosing either an RGB or HSV color space does not create significant differences, as both hues and luminances are well preserved. Computing DMDs for color images is three times slower than for grayscale images, given that we process each color channel independently.

Applications: We have illustrated our method with applications in image segmentation, simplification, and artistic manipulation. For all such use-cases, there exist obviously more specialized methods which yield better results. Our purpose in selecting these use-cases was mainly to illustrate the versatility of our framework, *i.e.* the fact that the proposed DMD representation can be seen as a potential, simple, alternative for a wide spectrum of image processing tasks. As such, we see the DMD as a low-level descriptor atop of which more advanced manipulations can be built, and not as an end-user instrument by itself.

4.6 CONCLUSIONS

We have presented dense medial descriptors, a new representation that encodes shape and luminance information in grayvalue images. To allow using medial descriptors for such images, we first decompose an image into all its possible threshold-sets, and then encode each such set using classical medial axes regularized by a corner-preserving saliency metric. The resulting descriptor allows an exact reconstruction of the initial image using distance-based interpolation techniques, and also an application-dependent simplification by eliminating shapes, or shape details, of low interest or relevance. We have implemented our descriptor using GPU-based techniques to achieve near-real-time performance. We demonstrate our proposal with applications in image simplification, segmentation, and artistic painting effects.

Many possible extensions of our proposal exist. First, we can exploit the topological information present in our dense medial axes, *e.g.* branching or looping structures, to perform higher-level image analysis tasks such as fuzzy object recognition. Secondly, we can exploit the spatial and topological relations of medial axes of consecutive image layers to perform new types of image editing, *e.g.* fuzzy object deformation, and also to study new methods for image compression. Finally, generalizing our method to 3D scalar volumes is an interesting avenue to explore.

4.6.1 *Ongoing work*

Following our initial proposal of the dense medial descriptors, several improvements and additions to the main idea have been proposed by others. Most notably, Terpstra [273] has presented a thorough study of the parameter space of the image compression proposed by us. This work shows that, by carefully setting the parameters of our pipeline (skeleton simplification, selection of relevant layers, run-length encoding of the layers), one can obtain compression rates of two to twelve times smaller than high-quality JPEG, while maintaining a similar (or slightly lower) image quality. In particular, the dense skeleton compression was found to be very effective

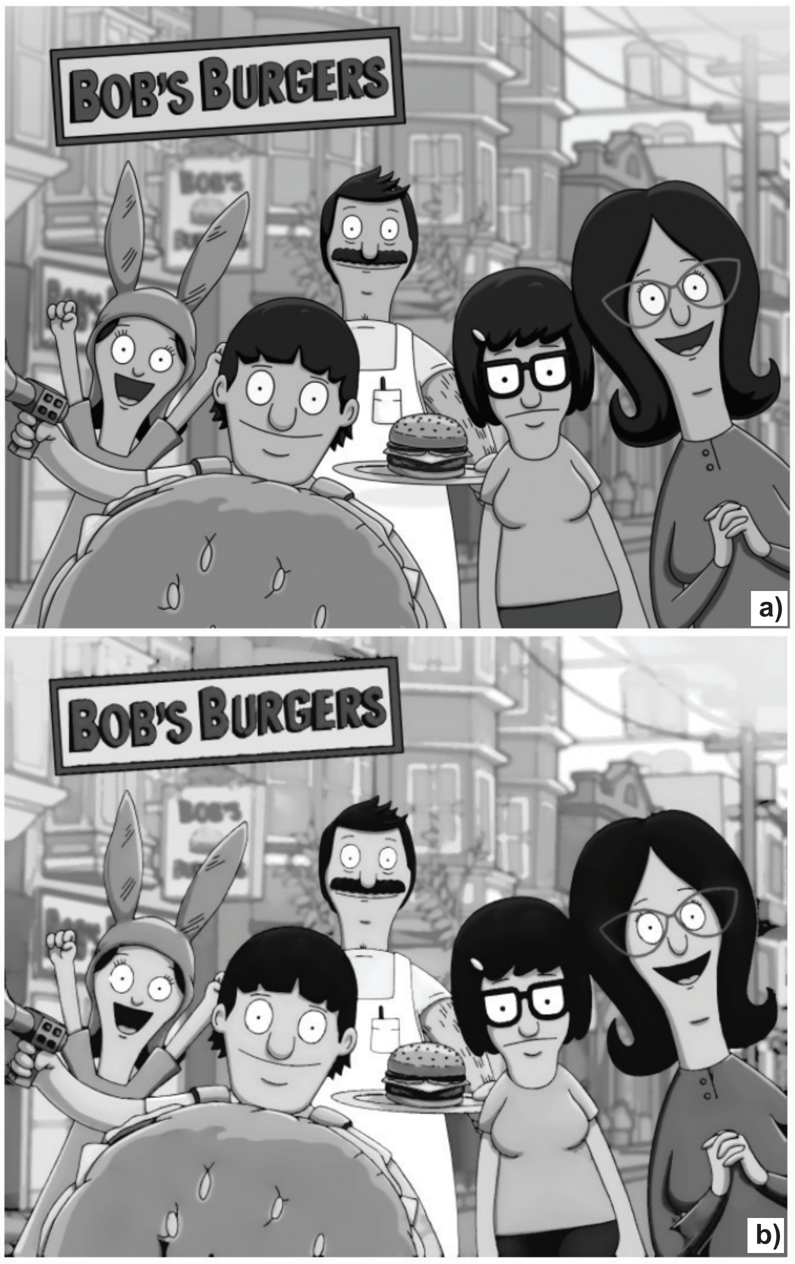


Figure 4.9: Dense skeletons used for image compression: (a) JPEG image (482KB, 1280×1014 pixels); (b) Same image, compressed with our method, yields a file which is only 184 KB in size. Image from [273].

for images which present few different intensities and many salient (high-contrast) and large shapes, such as design, cartoon-like, and other human-made images. Figure 4.9 shows such an example. Additionally, Terpstra has shown that our dense skeletons an be used as a preprocessor to increase the compression rate of standard JPEG by about 10% without visible quality loss, and also to postprocess images by supporting *e.g.* relighting. For full details, we refer to [273]. These results further strengthen our claims outlined in this chapter that dense skeletons are a useful image descriptor.

As explained in the introduction of this chapter, the initial driver of proposing a novel image segmentation method was to obtain an automatic method that would segment grayscale images captured from a video camera of an automatic milking device (AMD) so we can use the resulting segments to facilitate cow udder tracking. Due to limited time, we have not been able to validate this claim. Specifically, our dense-skeleton-based method has proven effective in delivering plausible segmentation of a variety of 2D grayscale and color images, as illustrated in this chapter. However, the method's parameters, *i.e.* the number of selected threshold sets and the degree of simplification of the computed skeletons, influence the resulting segments in subtle ways which do not, yet, allow us to claim that such segmentations are suitable for delivering us a separation of cow udders from the surrounding scene in a *fully* automatic manner. As such, while we maintain our claims that dense skeletons are a novel and interesting addition to the segmentation arena, more research is needed to fine-tune this method to make it fully suitable to the context of AMD tracking. Hence, in the next chapter, we will explore different methods that support our AMD tracking goal.

ACKNOWLEDGEMENTS

The work related to this chapter was co-financed by the research grant PN-II-RU-TE-2011-3-2049 offered by ANCS (National Authority of Scientific Research), Romania.

5 TRACKING COW TEATS

As outlined in Chapter 1, one of our two main research questions is how can visual analytics help understanding and improving the operation of automated, low-cost, computer vision tracking algorithms for 3D shapes from low-quality video data (**RQ1**). To answer this question, we need first and foremost a *use case*. This chapter provides such a use case in the form of designing a tracker to pilot a robot for the automatic milking of cows, in support of the dairy industry. We cover this goal by first presenting the context of our endeavor and its specific tracking constraints and requirements (Sec. 5.1). Next, we overview existing solutions, with a focus on methods that fit our use case's context (Sec. 5.2). Section 5.4 describes our tracker solution. Section 5.5 presents the obtained tracking results. Finally, Section 5.6 concludes the chapter[1].

The visual-analytics-based evaluation of the tracking results obtained with the method presented in this chapter, as well as how such analytics methods can be used to get insight and improve the tracker, forms the separate scope of Chapter 6.

5.1 USE CASE AND ITS CONTEXT

Scale economies in the dairy industry increasingly shift manual labor to robots. One such development is the advent of *automatic milking devices* (AMDs): Given a stable populated with cows, AMDs use vision devices to locate cows in the stable, reach under the cow *e.g.* with a mechanical arm, locate the udder and teats, and finally track the teats in order to couple a suction device to each teat to collect milk [111, 159, 178, 229, 290].

Vision devices used in AMDs must be small, shock-resistant, able to work in the dim lighting of a stable, and relatively cheap [290]. This already precludes the use of solutions such as stereo vision, which rely on delicately mechanically calibrated video cameras [111]. Separately, AMD vision devices have to operate in near-real-time to cope with the cow's motion, handle occlusions, locate features of interest with sub-centimeter precision, operate within a wide range of lighting conditions (including hazy images created by dust in the air), and work fully automatically.

In recent years, time-of-flight (ToF) range cameras have become increasingly popular as the core building-block of such AMD systems [178, 229].

1 The material in this chapter is based on the following publication: M. van der Zwan and A. Telea. Robust and fast teat detection and tracking in low-resolution videos for automatic milking devices. In J. Braz, S. Battiato, and F. Imai, editors, *Proceedings of the* 10^{th} *IEEE International Conference on Computer Vision Theory and Applications (VISAPP)*, volume 3, pages 654–667, 2015.

Given a 3D scene, a ToF camera produces a per-pixel depth map of the occluding surfaces found in front of the camera, with a relatively high frame-rate (24 frames per second (fps)). Compared to traditional stereo vision [111] or laser-scanning [110] devices, ToF cameras are less sensitive to lighting conditions and dust specks, generate a full depth-map with depth data at each pixel, are highly shock-resistant, come in compact form-factors, need no delicate calibration, and provide many 3D vision functions in embedded software [63, 64]. Hence, high hopes are placed on using ToF cameras in industrial AMD applications. However, their quite low spatial resolution (as compared, among others, to traditional stereo vision) creates new challenges that are not handled by mainstream computer vision algorithms.

In the following, we present a vision-based solution for AMD robots built using ToF cameras. We focus on the robust, accurate, automatic, and fast detection and tracking of cow teats, *i.e.*, the last step of the milking process. We present the entire pipeline from depth image acquisition, feature extraction and filtering, and udder tracking, and detail a simple and efficient implementation. We show both qualitative and quantitative validation of our system in an industrial context.

5.2 RELATED WORK

We next overview classes of computer vision methods for feature detection and tracking for natural deformable moving objects, such as cow teats. Given our application context, we focus only on methods which at least have some chance to comply with all our requirements: (1) automation, (2) low-cost, (3) robustness, (4) low computational complexity, and (5) implementation simplicity. Computer vision methods that obviously do not fit at least one of these requirements are excluded from the evaluation since they are not interesting in our context described in Sec. 5.1.

Marker-based tracking: A standard solution to 3D shape tracking is to mark salient keypoints thereof by textures which can be easily detected in a 2D image. If correspondences can be robustly found between stereo image pairs, stereo vision solutions can then be used to compute 3D positions of such fiducial marker-pairs by triangulation [145]. Marker-based solutions are fast, simple to implement, and quite robust, but not applicable to our context, as the dairy industry guidelines discourage the placement of markers on cow teats. Monocular marker-based tracking solutions also exist, but they are considerably more complex and computationally expensive for non-rigid, complicated, shapes [2, 248].

Image-based methods: Image-based solutions use the various 2D images delivered by a camera, *e.g.* luminance and depth, to detect salient image *features* that correspond to parts of the shapes to track. Such features can be, for example, corners, edges, local signal maxima, and edge crossings. These can be detected *e.g.* using SIFT [160] and SURF [19] descriptors.

However, very low-resolution texture-less images, like our cow udders, the robustness of SIFT and SURF approaches is very low. In the same class of image-based methods, *template*-based methods try to find pre-defined templates (small predefined patterns) in the image, using statistical approaches such as correlation [258]. Deformable dynamic templates (DDTs) can search for more complex configurations, by adapting a deformable template model to fit image silhouettes [307]. However, DDTs require well-chosen energy functions, initialization points, and high-resolution images, and are too computationally expensive for our real-time context.

3D reconstruction: Having a ToF camera, one can reconstruct the 3D visible-object surface from the depth field delivered by the ToF camera as a 3D point cloud. From this surface, teat tips could be, in principle, found at maxima of mean or Gaussian curvature, akin to polyp detection methods used in medical science, *e.g.* [42]. Yet, reconstructing clean, differentiable, 3D surfaces from point clouds given by ToF cameras is challenging. Most existing surface reconstruction methods have constraints on the sampling density, complexity, connectivity, and water-tightness of the sampled surface, and are also quite slow [57, 58, 109, 135, 141]. Also, such methods cannot find features (like our cow teats) which are occluded in the input image.

Specialized solutions: Many techniques have been proposed and fine-tuned to find and track features in moving natural shapes such as humans or parts thereof, *e.g.* faces or hands. However, such techniques are not directly usable for cow udder morphologies, as they have other shape priors. In the milk industry, very few solutions exist and have been implemented into AMD robots [111, 159, 178, 229, 290]. All these solutions assume a *fully* unoccluded *and* zoomed-in bottom or side view of the udder, given by a *fixed* robot arm that places the camera close to the udder, and given a cow constrained in a small space, to limit motion. In contrast, we do not assume that our robot is initially correctly placed close to the cow udder, nor do we assume that the cow cannot move *vs* the robot.

Given the above analysis, we exclude marker-based tracking and specific solutions from our range of interest, but keep 2D image-based tracking and 3D reconstruction as potentially viable solutions. In the next two sections, we describe how we explore the ‘tracker design space’ based on these two solution classes.

5.3 TECHNICAL SETUP

We first describe the technical setup used for data acquisition. As input device, we use a SwissRanger SR4000 ToF camera [177], which has one of the best quality-price ratios to the starting date (2010) of our research [63, 64]. The camera gives a 24-fps stream $\{I_i\}$. Each frame I_i has two 176×144

pixel images (A_i, D_i). A_i is a standard amplitude (luminance) image. D_i is a depth map, where each pixel stores the distance, in millimeters, to the closest occluding object, with an accuracy of a few millimeters for distances up to roughly 1 meter. Per frame, the camera also delivers a point-cloud $P_i = \{\mathbf{p}_j\}$ with the world-space locations of all visible-surface points in frame i.

The camera is rigidly mounted on a robot which can reach the zone under the cow to be milked. As outlined in Sec. 5.1, we focus on the milking stage, where the camera is already under the cow, roughly between the legs and looking towards the tail. The cow stands upright, so its legs and teats appear as vertically-oriented shapes in the image (see, for instance, Fig. 5.8a).

5.4 TRACKER DESIGN

As already detailed in Chapter 2, a typical shape-from-image tracker consists of a combination of a *detector* that finds relevant features (*e.g.*, shape parts) in the input image(s), and a *tracker* proper, that extracts the high-level evolution over time of the shape of interest by using the evolution over time of detected features. In our teat tracker design, we follow the same principle. Our solution has two parts: A *detection* step finds teats from the image-and-point-cloud $\{I_i, P_i\}$ of the current frame i. Next, a *tracking* step integrates this data over time, handling occlusion and other model priors. The interaction of these two components is depicted in Fig. 5.1, and detailed further in Sec. 5.4.1.3.

As already outlined in Chapter 2 and Sec. 5.2, many methods exist for both the detection and the tracking step outlined above. Examining all possible combinations is, we argue, a too large search space. As such, we propose to proceed in the tracker design by (a) examining a subset of methods which (at least) have chances to comply with the constraints of our application (type of shape to be tracked, noise level, and characteristics of the sensor device we use); and (b) excluding, in an early phase, methods which fail to deliver the required detection and/or tracking information. This analysis of the search space of possible methods is outlined next.

5.4.1 *Detection*

To find teats in the a frame I_i, we can use one or several of the fields A_i, D_i, and P_i given by the camera. After extensive studies, we found that our images A_i are very low-contrast and noisy, due to poor lighting in the stable. Hence, we use only the depth image D_i and point cloud P_i for teat detection. As D_i still contain a small noise amount, caused by dust specks floating in the stable, we first apply a median filter to them. The filtered images $\tilde{D}_i$ are almost noise-free and show little blurring (Fig. 5.8 b).

We next investigate four separate approaches to finding cow teats from 3D reconstructions of the point cloud (Sec. 5.4.1.1), segmented depth images (Sec. 5.4.1.2), template-based approaches (Sec. 5.4.1.3), and direct point-

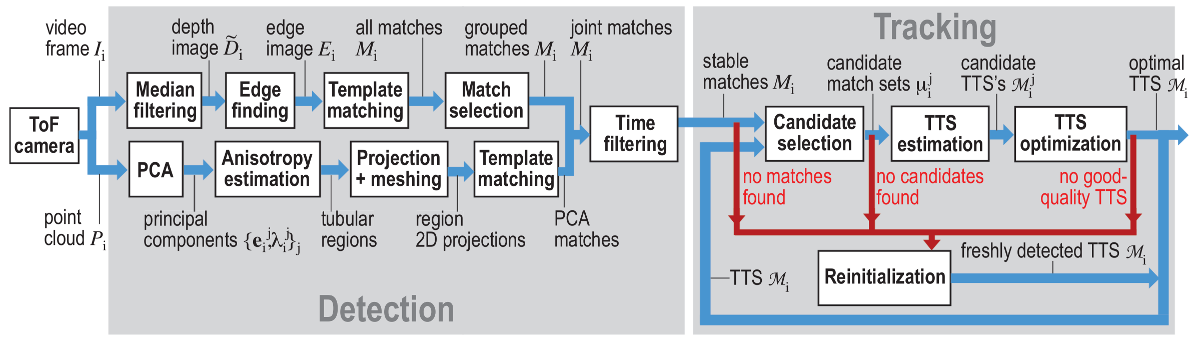


Figure 5.1: Teat detection-and-tracking pipeline. Blue arrows show data streams from the input data (ToF camera) to the output of four tracked teats $\mathcal{M}_i$. Red arrows show the control-flow for tracking reinitialization (Sec. 5.4.2.5).

cloud analysis (Sec. 5.4.1.4) respectively. The level of investigation of these approaches differs – some of the approaches are studied in depth, whereas others are discarded at an earlier stage. This is in line with the earlier-mentioned philosophy of giving a fair chance to existing method classes in the planned design, but excluding them from further study once the perceived ratio of cost (or complexity) *vs* benefits, exceeds what we deem feasible within the time constraints of our research.

5.4.1.1 *Detection: 3D shape reconstruction*

Our first detection approach, 3D shape reconstruction, takes the unstructured point set, or point cloud P_i, from the 3D ToF camera and tries to construct a surface $\mathcal{S} \subset \mathbb{R}^3$ that best approximates the points. The key idea here is that, once such a surface is available, one can use existing 3D shape analysis methods to locate protuberances such as teats, in a more effective way than when using the unstructured point cloud P_i alone.

Different methods are available in this area. For a survey of the literature, we refer to the very recent survey of Berger *et al.* [21]. The main idea of all these methods is to define an implicit or explicit surface $\mathcal{S}$ with constraints, and then compute the best fit between the surface and the point cloud P_i. Typical constraints include

- The local smoothness of the surface; Typically, locally smooth surfaces are preferred, which eliminates high-resolution, low-spatial-scale acquisition noise, and also enables the robust computation of subsequent differential metrics on the surface, such as *e.g.* curvature [43];
- The global topology of the surface. Depending on the method, simply connected, or multiply connected surfaces can be extracted. Other constraints include the water-tightness of the surface and its genus;
- The precision of the fit between $\mathcal{S}$ and P_i. At one extreme, we have interpolating surfaces which pass through the points, *i.e.*, $\mathbf{p}_j \in \mathcal{S} : \forall \mathbf{p}_j \in P_i$. At the other extreme, we have approximating surfaces which aim to minimize their closeness to P_i, typically described by means of a Hausdorff metric between $\mathcal{S}$ and P_i. Interpolating surfaces are preferred, but they can be very sensitive to outlier noise in P_i. Approximating surfaces deliver better smoothness and robustness to point-cloud noise, but they may have difficulties in capturing small-scale, but important, details in P_i.

We have tested five different well-known, state-of-the-art, methods for surface reconstruction from 3D point clouds on the point-cloud datasets delivered by the ToF camera. In all cases, we used the implementation provided by the authors. These methods are as follows:

- **PR:** Poisson-based surface reconstruction [135] as implemented in CGAL [38];

- **FFTPR:** Accelerated Poisson-based surface reconstruction using the Fast Fourier Transform [134];
- **PC:** the Power Crust method [5];
- **COC:** the Cocone method [56];
- **HPR:** hierarchical Poisson reconstruction (a fast and more robust extension of [135]).

For testing these methods, we proceeded conservatively in a two-step process, as follows.

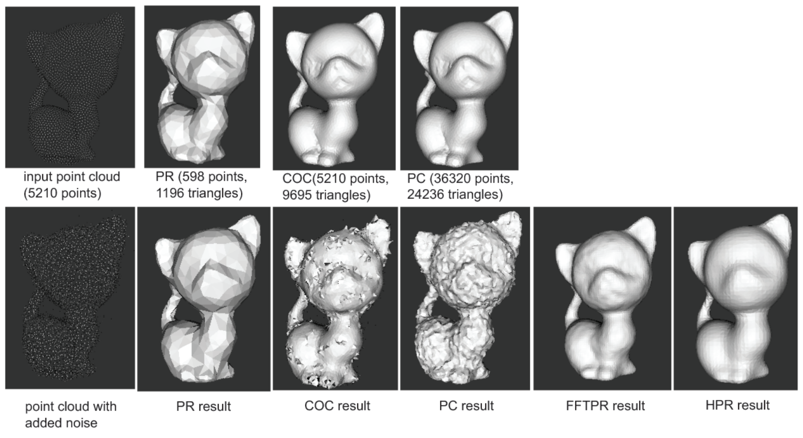

input point cloud (5210 points)
PR (598 points, 1196 triangles)
COC(5210 points, 9695 triangles)
PC (36320 points, 24236 triangles)
point cloud with added noise
PR result
COC result
PC result
FFTPR result
HPR result

Figure 5.2: Surface reconstruction from clean and noisy point clouds using five state-of-the-art reconstruction methods (Sec. 5.4.1.1).

First, we considered a number of relatively simple point clouds, obtained by retaining only the vertices of a set of closed, noise-free, compact, single-component, genus 0, and uniformly-sampled 3D mesh models. Such point clouds are arguably very simple – so, if a reconstruction method has issues here, it will be clearly unsuitable for our ToF point clouds which are noisy, contain multiple (usually non-watertight) objects, and have a variable sample density. This test servers thus as an ‘early culling’ of unsuitable methods.

Figure 5.2 shows the results of surface reconstruction on a simple 3D model processed by the five above-mentioned methods. The upper row shows the results obtained from the clean point cloud. The bottom row shows the results obtained from the point cloud to which we added random displacement noise to all points $\mathbf{p}_j$ of amplitude equal to 15% of the cloud diameter. As visible in the top rows, different reconstruction methods produce meshes of highly different resolutions – all being, however, of quite good quality. When noise is however added to the point cloud, the results are much more different – see bottom row. The COC and PC methods are quite sensitive to this noise, as they try to interpolate the point cloud.

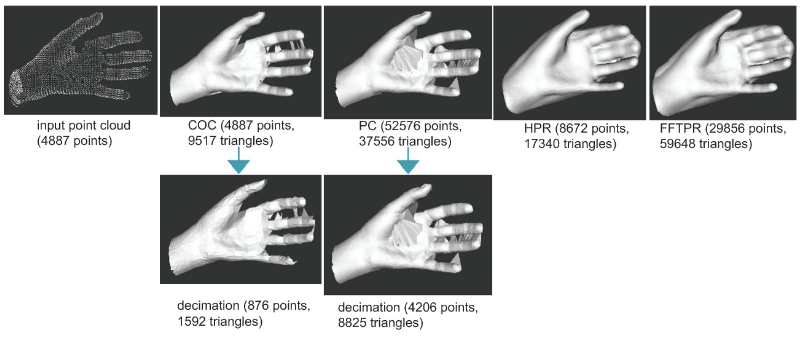

input point cloud (4887 points)
COC (4887 points, 9517 triangles)
PC (52576 points, 37556 triangles)
HPR (8672 points, 17340 triangles)
FFTPR (29856 points, 59648 triangles)
decimation (876 points, 1592 triangles)
decimation (4206 points, 8825 triangles)

Figure 5.3: Surface reconstruction from non-watertight point clouds using the same methods listed for Fig. 5.2 (Sec. 5.4.1.1).

The other methods (PR, HPR, and COC), which are based on approximation techniques, have a far less pronounced sensitivity to outlier noise.

We next refine the analysis by considering a more complex point cloud (Fig. 5.3, top row). While still coming from the vertices of a 3D polygonal mesh, this point cloud does not sample a watertight surface – the hand model has a large opening at the wrist and a smaller one between the index and the thumb. More importantly, the model has a number of protuberances which are both elongated and close to each other – similar to the teats of a cow udder. As shown by Fig. 5.3 (top row), the five considered method perform far worse than for the model in Fig. 5.2: All methods 'join' close fingers with spurious polygons, and the HPR and FFTPR methods also show a quite poor approximation (the reconstructed surface is inflated visibly beyond the point cloud around the wrist area). Another issue is that some of the considered methods create quite large meshes (tens of thousands of vertices and triangles). This is prone to cause performance issues for our subsequent shape-analysis operations, which, as mentioned in Sec. 5.1, should work in real time and with limited computational resources. To help this, such models can be decimated, using *e.g.* the method in [227], which is readily available in the well-known VTK toolkit implementation [228]. The bottom row of Fig. 5.3 shows two such decimations for the COC and PC reconstructions. While the mesh size is considerably reduced, the merging artifacts are still present. Such artifacts will immediately complicate most types of surface analysis for detecting protuberances.

From this analysis, which we have also performed on other point clouds (not shown here for the sake of brevity), we concluded that the only potentially useful method in this context is the hierarchical Poisson reconstruction (HPR) method. The HPR method is also interesting for our context as it does not require user input or scene-specific parameter settings. As such, we next test this method on real point clouds acquired with our ToF camera from cow udders. Figure 5.4 shows two such point clouds and the obtained reconstructions. The first point cloud (Fig. 5.4a) has been captured when the ToF camera was relatively close to the udder (about 50 centimeters),

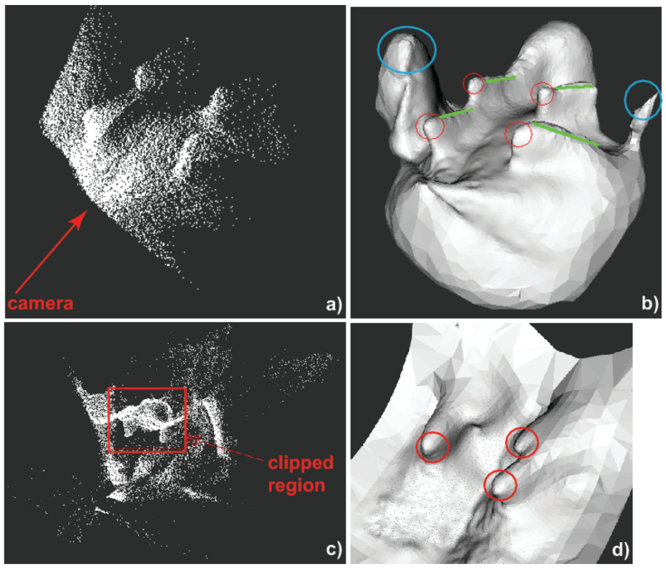


Figure 5.4: Surface reconstruction from actual cow-udder point clouds acquired with a ToF camera (Sec. 5.4.1.1).

pointing to the direction indicated by the red arrow in the figure. While the reconstruction (Fig. 5.4b) shows a relatively smooth and noise-free surface, it also has several drawbacks. First and foremost, we see that the reconstruction of the teat parts which are invisible from the camera's viewpoint (along the green lines emerging from the teat tips, indicated by red circles) is incorrect: Since there is no 3D point cloud information here, the only constraint the HPR method can use is that of surface smoothness. This creates artificially shallow slopes, or, in other words, teat shapes which are far wider than the typical rounded cylinders they should be. This can be a serious problem for further teat-tip detection methods that use the reconstructed surfaces, as such methods would typically assess the local curvature of the surface to detect strongly convex *and* small-scale shapes. If the reconstructed surface is smoother out by the HPR method due to unavailable 3D information, locating such tips can be unreliable. A second problem is that the HPR method computes, by construction, a closed (watertight) surface. This generates spurious protuberances, such as the two indicated in Fig. 5.4 by clue circles. These appear due to the limited aperture of the ToF camera, when the camera is close to the udder. Clearly, the geometry characteristics of these protuberances is very similar to that of the true teats (red circles), which makes their elimination challenging.

A second reconstruction test is shown in Fig. 5.4(c,d). Here, the camera is further from the udder (about 1 meter). The acquired point cloud is far more complex (and more non-uniformly sampled), as shown in Fig. 5.4c. Hence, more elements fall within the camera's frustum, such as parts of the animal's legs and belly. Reconstructing a *single* watertight surface from this point cloud delivers very large approximation errors. To further test reconstruction, we manually clipped the udder region from the 3D reconstructed mesh. The resulting shape (Fig. 5.4d is, again, relatively smooth (except, of course, along the clipping borders). However, due to the larger distance between the camera and the udder, distance-measurement errors occur, which result in teats which look relatively flat. More precisely, the clipped reconstruction shows only three teats (red circles in Fig. 5.4, of which the top-right one is very shallow). We conclude that the HPR method has serious challenges to handle our configuration.

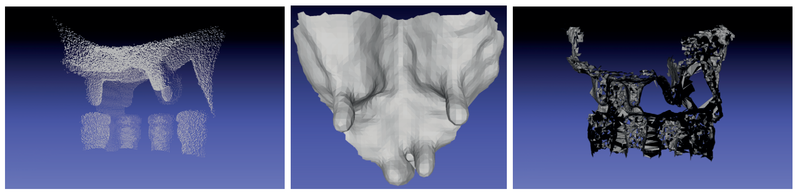


Figure 5.5: From left to right: Input point cloud, 3D reconstruction using the PR method [135], and reconstruction using the RIMLS method [192].

A third reconstruction test is shown in Fig. 5.5. Here, we select a point cloud which shows both the udder and the four suction cups of the AMD (Fig. 5.5, left). This cloud simulates a realistic scenario where the AMD would be very close to the udder and in the process of coupling the suction cups. This is a more complex geometry, as the point cloud contains several separated components. The PR method produces a relatively good result (Fig. 5.5, middle). However, to obtain this result, we needed, again, to manually clip the region of interest (udder and suction cups) from the total reconstructed mesh, to eliminate spurious parts. As a final test, we apply the RIMLS method [192] on the entire point cloud. The result, shown in Fig. 5.5 (right), has a clearly very poor quality.

Based on the insights obtained from the above experiments, we conclude that automatically reconstructing a clean udder surface, which captures well the geometry of the teats, is *extremely challenging* in general, even when using state-of-the-art methods. This is due to a combination of factors – high variability of the camera position *vs* the input geometry; complex nature of the input geometry, having several disconnected parts and significant occlusions; and limited resolution and accuracy of the ToF device. Given the above, we deem the path of detecting teats via 3D surface reconstruction to have a very low feasibility within our application context. For completeness, let us mention that other methods have recently entered the competition of 3D shape reconstruction from single luminance or depth images. Most notably, Jackson *et al.* have shown that it is possible to reconstruct quite accurate 3D models of human faces from single luminance or HSV images thereof [127]. In a different field, Bronstein *et al.* have shown that it is possible to reconstruct 3D watertight mesh models from highly in-complete single views of such shapes, taken with typical ToF devices [158]. While such methods work nearly fully automatically, and are quite fast (under one second on a modern computer), they require the training of complex deep neural networks, which in turn requires tens of thousands of ground-truth data samples. As we do not have such a number of labeled samples in our AMD context, we rule out this class of methods as well.

5.4.1.2 *Detection: 2D depth-image segmentation*

Our second approach uses the median-filtered depth images $\tilde{D}_i$. As outlined at the beginning of Sec. 5.4.1, these images are almost free of noise and have little blurring.

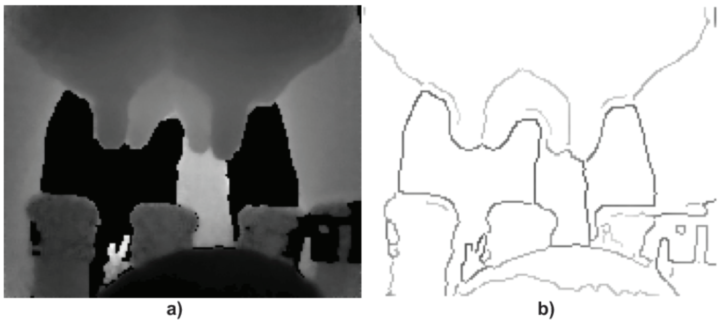


Figure 5.6: Depth image based detection of teats. Depth image (a) and its edges (b). See Sec. 5.4.1.2.

The key idea in the detection approach considered here is to apply image segmentation techniques to separate the udder from the surrounding objects. If this is reliably possible, then we can treat the segmented image part (udder) as a 2D shape and analyze it to find protuberances (teats). The idea is roughly similar to the approach proposed in Sec. 5.4.1.1, with the difference that we now work in 2D rather than in 3D. The added-value of the depth-map segmentation is that, if successful, this approach would have to process a 2D dense pixel image rather than a 3D sparse point cloud, which is arguably easier and computationally faster.

Figure 5.6a shows a typical depth map acquired with our ToF camera. Distance to camera is mapped to grayscale (close=darker, far=brighter); black is reserved to map pixels where depth estimation cannot be reliably done – typically because the first 3D shapes encountered along rays for these pixels are too far away for the ToF camera to measure a good signal. Figure 5.6b shows the results of an edge detector applied on the depth map, with edge strength mapped to grayscale (darker=stronger). We see that depth discontinuities separating teats and/or the udder from surrounding shapes are quite well detected, and the image does not show a large amount of edge noise.

We next test our proposal by applying several image segmentation methods known in the literature on the depth image. To this end, we have tried several classical image segmentation methods, such as normalized cuts [236], histogram thresholding [191, 277], active contours [133] using gradient vector flow [302], and mean shift [46]. Overall, none of the above methods has delivered good results in terms of accurate separation of the udder from its surroundings. Encountered issues range from oversegmentation, sensitivity to parameter setting, and undersegmentation in areas where the udder is close to other scene parts along the depth axis. Presenting the entire spectrum of found limitations is, in our view, not insightful or helpful towards our final goal. As such, we illustrate the typical encountered problems for a single method (mean shift) in Fig. 5.7. The figure shows three different depth images (A, B, C), segmented for two parameter configurations (top row *vs* bottom row). Segment borders are indicated in green. As visible, the results can be influenced quite heavily, in the sense of oversegmentation, by the parameter choices. Moreover, it is not easy to find a parameter configuration in which *all* teats are segmented. For instance, for frame A, the top-row shows quite limited oversegmentation (which is good), but only two teats are identified as segment borders (which is bad). Using other parameter settings, we can detect for frame A all four teats (see bottom row), but this creates considerable oversegmentation across the entire image (which is bad).

In this context, we have also tested our segmentation method based on dense skeletons which we have proposed in Chapter 4. While this method is able to produce overall plausible segmentations of quite complex natural images (see *e.g.* Figures 4.4 and 4.4.2), which compete successfully with mean shift, it has the same problems outlined above for segmenting depth maps acquired with our ToF camera as the other tested methods. In particular, oversegmentation is an issue; we have extensively explored the parameter space of the method (skeleton simplification, layer selection) but have not found configurations where the udder is reliably separated as a whole component from its surroundings. As such, while the dense skeletons method proved to be a good instrument for other types of segmentation problems, and also for other applications (*e.g.* image compression and simplification, see examples in Chapter 4), this method does not currently constitute a solution for our teat tracking problem.

Overall, the performed experiments indicate that depth-image segmentation is very hard to do automatically, at the right level of detail (thus, avoiding over- and undersegmentation), and in a way that guarantees that all teats are captured as segment border. Moreover, even if this were possible, what we actually would need for our application, is to segment the *entire udder* from its surroundings as a single, or at worst a very few, component(s). Indeed, if this is not possible, then we do not see how we can further reliably detect the *teats*. Consider, for instance, the images in Fig. 5.7: For the oversegmented ones, it is hardly possible to tell which if the resulting segments corresponds to a teat and which not. Given the above difficulties, we conclude that 2D depth-image segmentation, while a very attractive option from a complexity and computational efficiency perspective, is not the

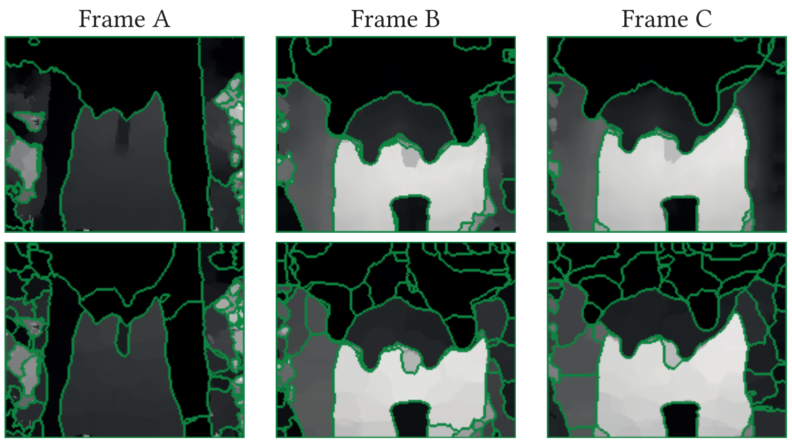


Figure 5.7: Applying mean shift segmentation to three different depth frames from a given video sequence. The top and bottom rows show results for two different parameter configurations.

best way to proceed. As such, we investigate next different solutions to this problem.

5.4.1.3 *Detection: 2D template-based solution*

Our first teat-detection method treats $\tilde{D}_i$ as regular grayscale images. To find teats, we use a template-matching technique consisting of four steps:

a. Edge detection: First, we find edges in the depth image $\tilde{D}_i$, using a gradient-magnitude filter $\|\nabla\tilde{D}_i\|$. The result E_i of this filter highights values where $\tilde{D}_i$ has strong jumps, which are the silhouettes of shapes in our depth image. Figure 5.8c shows a typical edge-image E_i. Silhouettes of the cow teats and limbs are clearly visible in this image.

b. Template matching: To find teats, we use a template-matching approach. For this, we first compute the silhouette (edge-image) of a typical U-shape of a teat. We call this image a template T (Fig. 5.8d). Next, we use a normalized correlation coefficient (NCC) approach [258] to find instances of T in the edge-image E_i, by convolving E_i with T using the Fast Fourier Transform provided by OpenCV [190]. Besides speed, the advantage of NCC becomes apparent if we notice that a teat could be close by in front of a leg, or far away from the background (stable wall), resulting in edges of highly different intensities. NCC efficiently corrects for edge-intensity differences in both E_i and T, which matches our goal to capture the *shape* of objects described by the edges, rather than objects' relative *positions* with respect to the background.

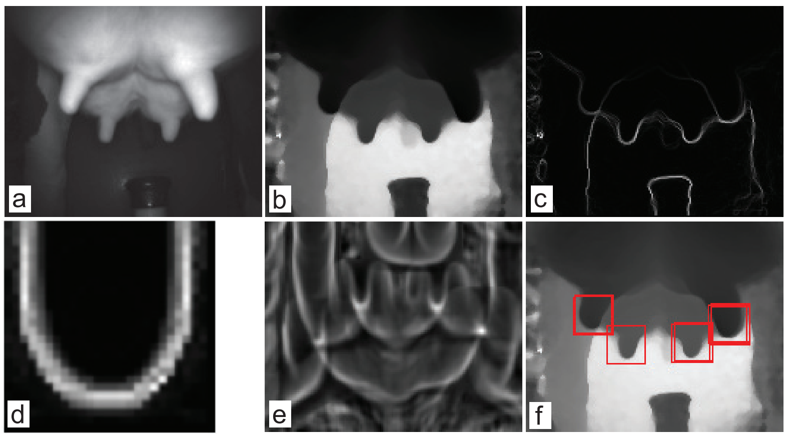


Figure 5.8: A frame from a typical video sequence. (a) Amplitude image A, with visible udder and four teats. (b) Filtered depth image $\tilde{D}$. (c) Edges E in depth image. (d) Canonical template image T. (e) Correlation image C_i. (f) Matches found (Sec. 5.4.1.3).

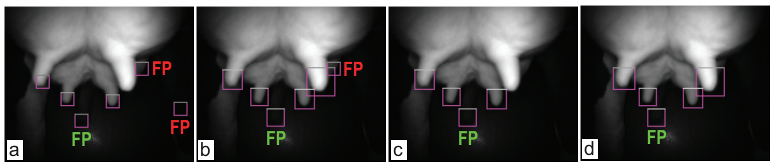


Figure 5.9: Single-scale (a) *vs* multiscale matching without time filtering (b). Multiscale matching with time filtering for two consecutive frames (c,d). Matches are indicated by rectangles, with *'FP'* showing false-positives. Red-marked FP's are removed by time filtering.

The NCC computation yields a correlation image C_i where each pixel $C_i(x, y) \in [0, 1]$ tells how well T matches the edge-image E_i at (x, y), with higher values encoding better matches (Fig. 5.8e). Maxima of C_i are regions where T matches best. Thus, we can find potential teat locations, or *matches* t_i, by finding the N largest local maxima of C_i. For all our tests, we fixed $N = 6$. We also tried the option of upper-thresholding C_i with a fixed value. However, this yielded between none and tens of matches per image C_i, so we prefer the first approach (N-best selection). For each match $t_i = (x, y, z)_i$, we store its 2D position (x_i, y_i) in image-space, and also its depth from camera z_i.

The above template matching method is not scale-invariant – it only finds areas in C_i which match the template T at T's own scale. Figure 5.9a shows this: Here, we miss the front-right teat, which is about twice larger than the template. Still, the range of teat sizes (in image-space) is bounded by the fixed size of the cow and the positioning of the robot which is never more than 1.5 meters away from the udder. Analyzing several production videos, we determined that teats range between 1/30 and 1/6 of the image-width, *i.e.* between $T_{min} = 10$ and $T_{max} = 30$ pixels. To find teats in this scale-range, we use the NCC method described above with six template sizes T_i, $1 \leq i \leq 6$, uniformly distributed between T_{min} and T_{max}. This enables us to find small and large teats (Fig. 5.9b).

c. Match selection: We next collect all matches t_i from all different scales T_j, after which we apply the N-best selection procedure outlined above for the single-scale case. When using multiple scales, we can find two (or more) matches t_i and t_j, for two scales T_a and T_b, whose 2D positions (x_i, y_i) and (x_j, y_j) are close enough to represent the same teat. We consider such matches to be duplicates when the center of the inscribed circle in T_i falls in the inscribed circle of T_j or vice versa (Fig. 5.10). From any set of duplicates, we only keep a single match for further processing.

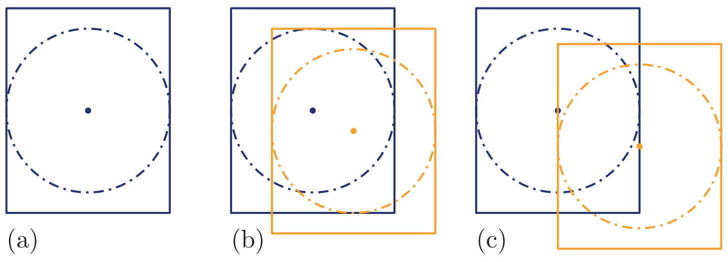


Figure 5.10: Template overlap. (a) Canonical template, with its inscribed circle and circle-center. (b) Two overlapping templates. (c) Two non-overlapping templates (see Sec. 5.4.1.3).

d. Match time filtering: Our teat-detection can find a teat where none actually exists. These are areas where the edge-structure in E_i has U-shapes similar to our templates, *e.g.* around the cow's tail-tip, or around some leg muscle structures. We call these *false positives* (FPs). Many such FPs appear only for a very few consecutive frames. In contrast, *true positives* (TPs) are visible for longer periods, until they get occluded or drift out of the camera view. We remove FPs by time filtering, as follows. Let $M_i = \{t_j\}$ be the set of matches found in frame i of our input stream. Given the sequence $\{M_k\}_{i-K<k<i}$ of matches found in the previous K frames, we remove from M_i those matches which are not visible in at least τ of the last K frames. This means that we have a delay (of K frames) in detecting teats. Choosing a low value for K keeps this delay small, as our camera operates at 24 fps. Fixing $K = 5$ and $\tau = 2$ frames effectively removed most FPs while keeping most TPs. Figure 5.9 shows this. The three FP matches marked red in images (a,b) are removed in image (c) by time filtering. The remaining FP, marked green, which corresponds to the cow tail, is however not removed, as this structure persists in several frames. We show next in Sec. 5.4.2 how such remaining FPs are removed by using tracking.

5.4.1.4 *Detection: PCA Based Detection*

The template-based method described above works well when teats are roughly vertical and parallel to the camera plane, *i.e.*, when the angle α between a teat's symmetry-axis and the camera plane is below roughly 10°. For such angles, the difference between the edge profiles of the vertically-aligned U structures in our templates T_i and those of actual teats in E_i is small enough to yield strong matches.

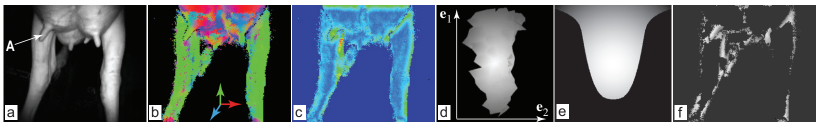


Figure 5.11: PCA-based detection. (a) Amplitude image. (b) Major eigenvector direction. (c) Elongation values. (d) 2D projected neighborhood of point *'A'* in the first image. (e) Template used for matching. (f) Correlation image (Sec. 5.4.1.4).

For larger angles α, template matching has difficulties. In such cases, the teats' silhouettes in E_i differ too much from the ones in our templates. We find two sub-cases here. First, a teat could be rotated *into* the camera-plane. To address this, we could use a solution akin to the one dealing with scale-variance (Sec. 5.4.1.3), *i.e.*, create a family of templates T_i^{rot} rotated in the camera plane. The second case occurs when teats are rotated *out* of the camera plane (see *e.g.* the two front teats in Fig. 5.11a). In such cases, the teat silhouette changes from a U-shape to an ellipse or parabola sector. We verified that rotation invariance cannot be dealt with in this case by using additional templates, as such shapes have too high an edge variability in the depth image.

We propose next a method to handle both rotation variance cases. Teats have a roughly cylindrical shape, which means that locally there is a clearly-oriented structure in the depth-image data. This structure can be lost in the projected edge image. To find such structures, consider a ball B of fixed radius, roughly 4 cm in world space, corresponding to the average half-length of a cow teat. We next center B consecutively at all locations $\mathbf{p}_i$ of the point cloud P_i delivered by the ToF camera, and compute the eigenvectors $\mathbf{e}_i^j$, $1 \leq j \leq 3$, and corresponding eigenvalues $\lambda_i^1 \geq \lambda_i^2 \geq \lambda_i^3$ of the covariance matrix of all points in $P_i \cap B$. Figure 5.11b illustrates this, by showing the direction of the major eigenvector $\mathbf{e}_i^1$ by color coding – red, green, and blue show eigenvectors $\mathbf{e}_i^1$ aligned with the x, y, and z axes respectively. Next, we find tube-shaped regions $P_i \cap B$ by computing the so-called linear anisotropy or elongation $c = \lambda_i^1/\lambda_i^2$ [291], and selecting only regions for which $c > 1.5$. These are potential teat locations. Figure 5.11c shows the elongation c with a rainbow colormap (blue=low, green=medium, red=high values). As visible, areas around teats are green, as they have a quite high elongation. Finally, we project such regions onto the plane defined by $(\mathbf{e}_i^1, \mathbf{e}_i^2)$. If a teat exists around $\mathbf{p}_i$, $\mathbf{e}_i^1$ should match its symmetry axis (given the teat's cylin-

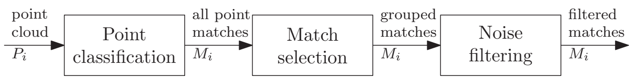


Figure 5.12: Pipeline of the 3-D template matching method, input moves from left to right (Sec. 5.4.1.5).

drical shape), so the resulting 2D projection should show a vertical teat shape, like the ones in our templates. This corrects for the rotational variance. Additionally, we scale the 2D projection by the value of λ_i^1 divided by the height of the template T, which takes care of the scale variance. As such, we can now directly use our *single-scale* template matching to find rotationally-invariant teat matches in the projected images.

Given camera resolution limitations, the 2D projections of cloud points $P_i \cap B$ can yield very sparse point sets. To match these with a teat shape, we need a compact image. To create this, we render a quad mesh with points $P_i \cap B$ as vertices and connectivity given by the raster structure of I_i. Mesh vertices are colored by their depth to the projection plane. Figure 5.11d shows such a 2D projection for the neighborhood $P_i \cap A$ around point A in Fig. 5.11a. Such images typically have jagged edges, given (again) the low resolution of our cloud P_i clipped by the ball B. Computing edges on such images yields a high amount of noise, which makes our edge-template matching not robust. We solve this by a template matching using the *full* image of a teat, where pixel grayscale values indicate depth (Fig. 5.11e). The correlation result (Fig. 5.11f) emphasizes elongated regions whose maxima correctly capture positions of rotated teats.

Matches found by PCA detection are merged with the ones given by the template-based detection (Sec. 5.4.1.3) to yield the final match-set M_i. This way, we increase the chances of capturing all matches in a single image. We next use this joint match-set M_i to robustly detect and track all four teats.

5.4.1.5 *Detection: 3D template based solution*

In the previous section we have seen that the point cloud P_i produced by the time-of-flight camera can be used as an alternative to the depth image D_i as source of input when detecting teats. However, the used process is quite involved and the projection step requires a significant amount of computations and, therefor, time. In this section we present a different approach to the detection of teats in the point cloud P_i produced by a time-of-flight camera. This approach is based on a simple 3-dimensional template matching approach which despite its simplicity proves very efficient for this purpose.

The 3-D template matching method consists of a number of steps which are similar to those used by the 2-D template method. Figure 5.12 shows the pipeline of going from the point cloud P_i to the (filtered) set of matched teats M_i, comparing this to the 2-D template detection part of Figure 5.1 we mostly see a difference in the type of data, stemming from a difference in

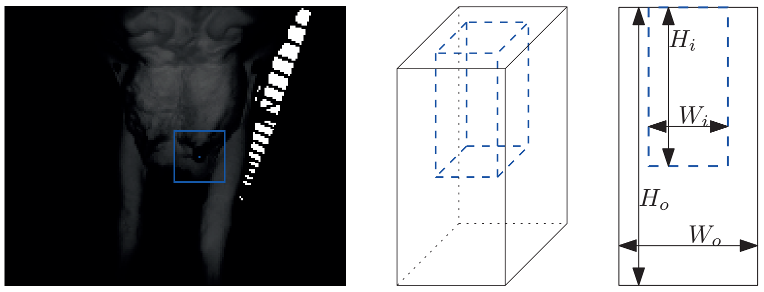


Figure 5.13: The moving parts of the 3D template detector, from left to right: The image window overlaid on the amplitude image, a schematic of the 3D template showing the inner and outer template, and a side view of the 3D template with important measures (Sec. 5.4.1.5).

input data. In the remainder of this section, we will provide details for each step.

POINT CLASSIFICATION The first step of our 3-D teat detection method is to mark all points in the point cloud as being part of a teat or not. In order to be able to go through the point cloud and look at the neighborhood of a point, we need some sort of connectivity information. While the time-of-flight camera we used does not produce a connected point cloud, we can use the fact the point cloud P_i is constructed from the images I_i, as we did in subsubsection 5.4.1.4. Using the connectivity and overall structure of the pixels in I_i we have a structured way of going through the corresponding points in P_i and determining their neighbors.

In Figure 5.13 we see a typical template T and its dimensions as used by our detection method. As can be seen in this figure, the template T consists of an inner template T_i and an outer template T_o. In general, we choose T_o such that is a uniformly scaled up version of T_i, aligning the top parts of both templates. An alternative shape we have also experimented with is that of a cylinder, using similar ratios between the inner and outer template as we did for the box shaped template.

Using the structure of image I_i we move the template T through the point cloud P_i such that the point to consider p is in the center of T_i. We then count the number of points inside both templates T_o and T_i, excluding points inside T_i from the count for T_o. By selecting an inner template size that corresponds to the size of an average teat, we can expect a high number of points inside the inner template when the template is centered on a point that is part of a teat while the number of points outside should be relatively low. On the other hand, when we compute N_i and N_o for a point that is part of the leg, we will again find a high N_i, but also a high (or at least higher) N_o, because there will also be part of the leg inside the outer template.

Considering the above, we can use the ratio between N_i and N_o to determine if the considered point p is part of a teat. Therefore, we define the function $\theta(p) = \frac{N_i}{N_o}$ as this ratio between point counts. Theoretically, this ratio could become infinite when N_o is zero. However, because the teats are close to other parts of the udder (or to each other) this hardly occurs in practice. At the same time, when we find a significant amount of points inside the inner template and none inside the outer template, we have the cleanest match for a teats that is possible. Therefore, we set $\theta(p) = N_i$ when N_o is zero making sure we never get a division by zero and still get a useful value for θ.

As stated above, we want there to be a significant number of points inside the inner template in order to consider the point p part of a teat. Our experiments have shown that a good minimum for N_i is 20 points, which can still be achieved when the teat is at a large distance from the camera. For the ratio between inner and outer points, we found that $\theta > 2$ yields good results.

MATCH SELECTION In contrast to the template matching method used by our 2-D template detector, the 3-D template matching technique described above produces a *binary* mask instead of a correlation map. In both cases, however, we have multiple points (or pixels) belonging to the same physical teat. We have therefore extended our 2-D template match merging algorithm to work for our dual 3-D templates, as follows.

In order to determine if two points p, q, which are identified as belonging to a teat, belong to the *same* teat, we look at the intersection of inner templates centered on the points p and q. If $T_i(p)$ and $T_i(q)$ intersect, we say the points p, q do belong to the same teat. Conversely, if the templates $T_i(p)$ and $T_i(q)$ do not intersect, the points belong to different teats.

Applying the process described above to all points marked as possible teats reduces the (regions of) marked points to a collection of teat positions in 3D. The resulting set of teats is remarkably stable over time and is better able to detect teats in situations which are challenging for the 2D template detection method, such as teats that are close to legs (from the camera's point of view) or teats that are almost completely hidden behind another teat.

NOISE FILTERING The 3D template detection method is, as described above, quite capable at detecting teats in all kinds of configurations. However, not unlike the 2D template detection method, it sometimes flags image parts as being a teat while there are not. Keeping in mind that the next stage, the tracking model, should be able to deal with such issues, we would still like to reduce the number of such so-called *false positives*.

The most important cause of false positives we experienced during our testing is caused, even after applying a noise reduction filter, by noisy input points. Therefore, we added an extra step to our 3D template detection method to reduce the occurrence of these noise-based false positives.

Observing the characteristics of the false positives based on noisy points, we found that the point ratio θ for these points can be anywhere within the

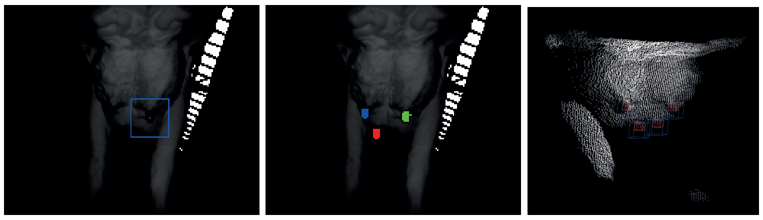


Figure 5.14: Example of using the 3-D template matching based technique for detection teats. From left to right: Amplitude image showing the search region around a point. Marked teats as found by the detector, all colored pixels correspond to teats which were classifieds as part of a teat. A point cloud showing the teats and matched 3-D templates (Sec. 5.4.1.5).

range of θ for true positives. Therefore, it is not possible to reject a detected teat based on the point ratio θ solely. On the other hand, the way the pixels corresponding to the points inside the template are laid out in the image is very different for false positives than for true positives. For a true positive, the pixels create a small and dense clump, whereas the pixels corresponding to a false positive occupy a bigger area and are spread in a more sparse way over the image. Therefore, we can find the difference between false and true positives by computing the sparsity of the set of pixels corresponding to the points inside the template. If we find this value too low (too sparse pixels), we reject the detected teat, or in other words we flag it as a false positive.

Due to the design of our detection and tracking pipeline, we can use the 3D template based detector instead of the 2D template based detector with little effort. In subsection 6.2.2 we will compare the difference in detection performance between the two detectors.

5.4.2 *Tracking*

Our teat detection technique (Sec. 5.4.1) successfully finds about 90% of the visible teat tips in our typical videos. Yet, detection still suffers from two main problems:

Occlusion: In frames where one or more teats are occluded from the camera viewpoint (by cow limbs, other teats or robot parts), detection obviously fails to find such teats. As our AMD robot needs to find *all* teats in each frame to start the milking process, we must locate occluded teats too.

Robustness: Even for frames with no apparent teat occlusion, two additional teat detection problems exist. First, certain teat configurations are not detectable, due to resolution limitations of the ToF camera. We call these *false negatives* (FNs). Some FNs can be removed by relaxing the detection method's parameters, to accept more image structures as teats. However, this makes detection sensitive to small-scale noise, which next creates matches at spurious image locations, *i.e.*, yields unwanted *false positives* (FPs).

To reduce the amount of FPs and FNs described above, we need to use additional information not present in single video frames. For this, we choose a *model-based* approach: We define a parameterized model that describes the intrinsic variability (priors) of shape, size, orientation, and dynamics (change in time) of the *entire* set of four teats that a typical cow has. At frame i, this set of teats, called the tracked teat-set (TTS), is a quadrilateral $\mathcal{M}_i = \{\mathbf{p}_j \in \mathbb{R}^3\}, 1 \leq j \leq 4$, whose vertices $\mathbf{p}_j$ are ordered counterclockwise with $\mathbf{p}_0$ being the near-left teat from the camera viewpoint. To compute $\mathcal{M}_i$, we use a *tracking* procedure that fits the TTS $\mathcal{M}_{i-1}$ computed from frame $i-1$ to the match-set M_i detected in the current frame i, subject to our model's geometric and dynamic constraints. Figure 5.15 shows the TTS quad tracked in three frames in a video of several minutes. Our tracking proposal is detailed next.

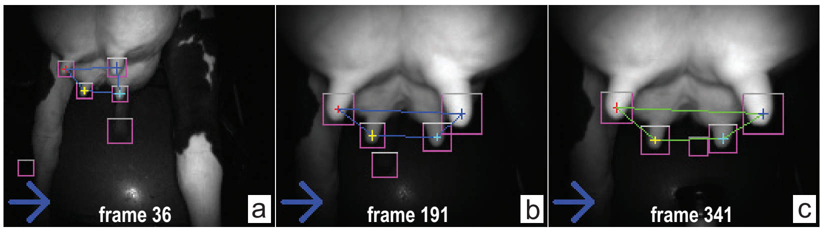


Figure 5.15: Three frames from a tracking sequence with matches shown as rectangles and TTS shown as a 3D quad (see Sec. 5.4.2).

5.4.2.1 *Candidate matches*

Key to tracking is finding how vertices of the TTS $\mathcal{M}_{i-1}$ from the previous frame correspond to teat-matches in M_i found in the current frame. To find these correspondences, we first construct a collection $S = \{\mu_i^j\}_j$ of all *candidate-match sets* $\mu_i^j \subset M_i$ each having between one and four matches as elements. We sort this sequence decreasingly on the number of elements $|\mu_i^j|$ in each candidate-match set (CMS), and then try to construct a candidate TTS $\mathcal{M}_i^j$ from each such μ_i^j, in increasing j order. This ordering models our preference to fit our TTS to more, rather than to fewer, matches in the current frame, so as to use most of the information present in that frame.

5.4.2.2 *Correspondence finding*

Given a CMS μ_i^j, we find its point-to-point correspondence with the previous TTS $\mathcal{M}_{i-1}$ as the set of point-pairs $\{(\mathbf{q}_k \in \mu_i^j, \mathbf{p}_k^{i-1} \in \mathcal{M}_{i-1})\}$, $1 \leq k \leq |\mu_i^j|$, which minimize the metric

$$E_{motion} = \frac{1}{|\mu_i^j|} \sum_{k=0}^{|\mu_i^j|} \|\mathbf{q}_k - \mathbf{p}_k^{i-1}\|, \tag{5.1}$$

where $\|\cdot\|$ is the Euclidean distance in $\mathbb{R}^3$. Intuitively, E_{motion} captures the amount of motion between $\mathcal{M}_{i-1}$ and $\mathcal{M}_i$. Since the cow stays relatively still during milking, the robot moves slowly, and our camera has a high frame-rate, teats cannot 'jump' from one place to another one between consecutive frames. Hence, for a CMS μ_i^j to be valid, it has to yield a small value for E_{motion}. In practice, we allow only values $E_{motion} < 25$ mm.

5.4.2.3 *TTS estimation*

From each CMS μ_i^j given by correspondence finding, we build a potential new TTS $\mathcal{M}_i^j$ for the current frame i: For all points $\mathbf{q}_k \in \mu_i^j$ which have a correspondence to a TTS-quad vertex $\mathbf{p}_k^{i-1} \in \mathcal{M}_{i-1}$, we set the new value of $\mathbf{p}_k^i \in \mathcal{M}_i$ to $\mathbf{q}_k$. For all other vertices $\mathbf{p}_k^i \in \mathcal{M}_i^j$ which have no correspondences in μ_i^j, a situation which occurs when $|\mu_i^j| < 4$, we compute their values by translating their corresponding points $\mathbf{p}_k^{i-1} \in \mathcal{M}_{i-1}$ with the average translation vector

$$\mathbf{v} = \frac{1}{|\mu_i^j|} \sum_{k=0}^{|\mu_i^j|} \mathbf{q}_k - \mathbf{p}_k.$$

5.4.2.4 *TTS optimization*

The previous step delivers as many potential TTS models $\mathcal{M}_i^j$ as the number $\|S\|$ of CMS configurations. These are all possible TTS models which can be built by using one or several matches in M_i. We select the best such TTS as the optimal TTS with respect to three metrics which describe geometric constraints observed by watching videos of actual cows during milking, as described below. Let us stress here that we are not searching for an absolute minimum of these metrics, but for a 'best fit', *i.e.*, a TTS which optimizes these metrics over all possible TTSs.

Shape: During milking, the soft udder shape changes as the cow moves. Yet, the *relative* teat positions are quite stable. Thus, the *shape* of our quad $\mathcal{M}_i^j$ should be constrained. While this is partly done by the motion constraint E_{motion}, that allows teats to move only slightly, an accumulation of such small movements over hundreds of frames can yield very different quad shapes. We thus further constrain the quad shape by constraining its area. We could have used other shape metrics here, *e.g.* the quad's aspect ratio. However, the area constraint performs much better during the tracking-initialization stage (see next Sec. 5.4.2.5). We model the area constraint by the difference between the actual quad-area and the expected quad-area $A_{expected}$ as

$$E_{shape} = \frac{|A(\mathcal{M}_i^j) - A_{expected}|}{A_{expected}}. \tag{5.2}$$

Here, $A_{expected}$ is a fixed value, computed from actual udder measurements of the cows under analysis. Setting $A_{expected}$ has to be done only once, before the first time the cow is milked, and can be re-used for subsequent milking.

Flatness: We also observed that teat tips stay roughly in the same plane. We therefore want the same to hold for the vertices of the quad $\mathcal{M}_i^j$. We model this by checking how close each vertex $\mathbf{p}_k \in \mathcal{M}_i^j$ is to the plane formed by the other three vertices, *i.e.* by the metric

$$E_{flatness} = \frac{1}{4}\sum_{k=0}^{4} |\mathbf{n}_k \cdot \mathbf{v}_k|. \tag{5.3}$$

Here, $\mathbf{n}_k$ is the normal of the plane through all quad points except $\mathbf{p}_k$, and $\mathbf{v}_k$ is the normalized vector from any point $\mathbf{p}_{l\neq k}$ to $\mathbf{p}_k$. When our quad is flat, every $\mathbf{p}_k$ lies in the same plane as the other points $\mathbf{p}_{l\neq k}$, so $\mathbf{n}_k$ and $\mathbf{v}_k$ are orthogonal to each other, thus $E_{flatness} = 0$. Higher values of $E_{flatness} > 0$ tell that $\mathbf{p}_k$ do not all lie in the same plane. In particular, note that configurations that include an incorrectly detected point on the cow's tail yield a high $E_{flatness}$, thus are not favored by this metric.

Orientation: Finally, we note that teat tips are in a plane roughly parallel to the ground surface on which the cow stands. We encode this prior by measuring the orientation-deviation between the quad vertex-normals $\mathbf{n}_k$, computed as for the flatness criterion, and the vertical direction $\mathbf{u}$, by

$$E_{orient} = \frac{1}{4}\sum_{k=0}^{4} |1 - \mathbf{n}_k \cdot \mathbf{u}|. \tag{5.4}$$

In the ideal case, all normals $\mathbf{n}_k$ are parallel to $\mathbf{u}$, so $E_{orient} = 0$. Values $E_{orient} > 0$ indicate deviations from the desired orientation. Similar to the flatness metric, the orientation metric typically produces higher values for incorrectly oriented vertices and therefore also favors the correctly oriented configurations, even when the corresponding value for E_{orient} is not optimal in an absolute sense.

To jointly optimize for TTS shape, flatness, and orientation, we use the total geometric error

$$E_{geom} = w_{shape} \cdot E_{shape} + w_{flatness} \cdot E_{flatness} + w_{orient} \cdot E_{orient} \tag{5.5}$$

where the weights w sum up to 1. The first TTS $\mathcal{M}_i^j$, in the testing order given by CMS finding (Sec. 5.4.2.1), that scores $E_{total} < \epsilon$, is considered a good-enough fit, and yields the new value for the TTS $\mathcal{M}_i$ for the current frame i. Here, we use $\epsilon = \frac{1}{3}$, meaning that only one of the three error metrics can be at its acceptable maximum, while all other error metrics should be zero for us to accept this configuration.

5.4.2.5 *Initialization*

To start tracking, we must initialize our TTS $\mathcal{M}$. Also, re-initialization is needed when we cannot track $\mathcal{M}_{i-1}$ to the current frame i. This happens when (a) the current match-set M_i is empty, *e.g.* due to a bad camera angle, too large distance to the cow, complete occlusion of teats in frame i, or limitations of our teat-detection algorithm; (b) no correspondence between $\mathcal{M}_{i-1}$ and M_i exists which satisfies the motion constraint E_{motion} (Sec. 5.4.2.2), *e.g.* because of accidental robot jumps due to collisions with the cow; (c) no candidate TTS $\mathcal{M}_i^j$ having a sufficiently good geometry E_{geom} is found, *e.g.* due to the same reasons as for (a).

In all such cases, we must build $\mathcal{M}_i$ afresh, using only data from M_i. For this, we first find all CMS sets μ_i^j having *at least* three points, by the same method as for tracking (Sec. 5.4.2.2). We regard each μ_i^j as a potential TTS $\mathcal{M}_i^j$, and compute its E_{geom}. The TTS yielding a minimal E_{geom} value below our threshold ϵ becomes our new $\mathcal{M}_i$. If no such TTS is found, we set $\mathcal{M}_i = \emptyset$, *i.e.* mark that tracking is lost in the current frame, and try to re-initialize in the next frame.

Let us further detail the difference between tracking and initialization. During tracking, we optimize for the TTS that (a) fits the most matches found in the current frame, (b) has the best geometric quality, and (c) has a small motion with respect to the previous TTS. In contrast, at initialization we only optimize for geometric quality and number of matches. Indeed, we cannot optimize for motion, since the previous valid TTS may have occurred many frames ago or there was no such TTS (at the video stream start). To track, we only need a *single* valid match in each frame. For initialization, we need minimally *three* valid matches in a frame (to be able to evaluate the geometric constraints). As we shall see in Chapter 6, our tracking is robust enough to require re-initialization only very seldomly, and thus deliver a high overall quality of the proposed solution.

5.5 RESULTS

Our tracking-and-detection system, implemented in unoptimized C#, achieves tracking at 4…8 fps on a 3.0 GHz Windows PC for an input video stream provided by the SR4000 camera API. For an image resolution of N pixels, both computational and memory complexities of detection are $O(N)$; for tracking, these are both $O(1)$, since the match-set sizes are not a function of the image size, but of the anatomical complexity of the udder. This strongly suggests that an optimized implementation, *e.g.* in embedded (parallelized) C, can run at real-time rates on a low-cost ARM processor such as available on the milking robot, which further supports our claims for practical industrial applicability and low cost.

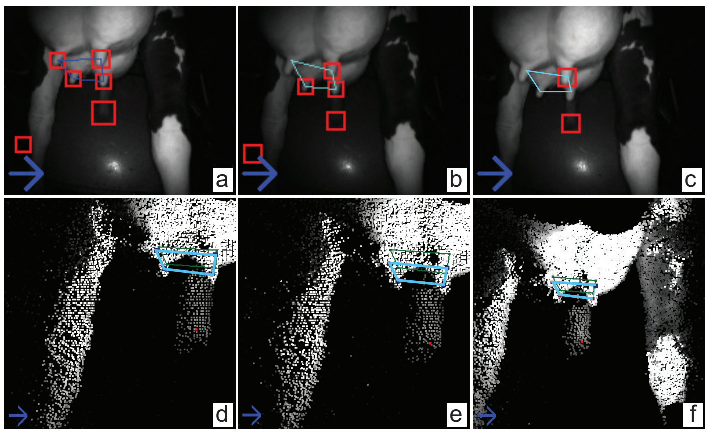


Figure 5.16: Tracking sequence, 3 consecutive frames. Top row: amplitude images, with matches shown. Bottom row: zoom-in on the point cloud around the tracked TTS. The blue arrow icon shows that the system is successfully tracking (Sec. 5.5).

Figure 5.16 shows the interaction between detection and tracking by showing the TTS results for 3 sequential frames selected from a longer video. The first frame (a) is an initialization frame. Here, five matches are found (red rectangles). Of these, the correct four corresponding to teats are selected by the initialization procedure (Sec. 5.4.2.5) to create the current TTS $\mathcal{M}_a$, as using any of the other two false-positives would create tilted quads which yield a high error E_{geom}. The obtained TTS is shown in Fig. 5.16d atop of a rendering of the point cloud zoomed in on the udder area. As can be seen, the TTS approximates the actual teat positions quite well. In the second frame (Fig. 5.16b), we find only three true-positive matches on the teats, and two false-positives. However, as seen in the corresponding cloud rendering (Fig. 5.16e), tracking correctly estimates the position of the fourth teat. In the final image, we only detect one true-positive and one false-positive (Fig. 5.16c). Here again, the tracking succeeds in creating the correct TTS (Fig. 5.16f).

5.6 CONCLUSION

In this chapter, we have presented the design and implementation of an end-to-end tracker system for the detection of cow teats for automatic milking devices (AMDs) in the milk industry. We present several techniques and algorithms that make this detection robust and fully automated when using a very low resolution time-of-flight camera, which renders classical computer vision algorithms not applicable. By combining depth and point cloud information analysis with observed model priors, we achieve a simple and robust implementation that can successfully track over 90% of the frames present in typical AMD videos, which exceeds the performance of all competitive solutions in the area that we are aware of. In contrast to these solutions, our proposal is also fully automated, allows large relative camera-subject motions and orientation changes, and accounts for occlusions.

Several observations are relevant with respect to the work presented here, as follows.

DESIGN SPACE: The design space of such an AMD tracker is, clearly, very large, and encompasses choices regarding the type of information to use (luminance image, depth image, or both); type of techniques used to extract the relevant features (teats) we are interested in (3D shape reconstruction followed by 3D shape analysis, 2D image segmentation, 2D template-based detection, and 3D template-based detection); and type of tracker to be used (individual teat *vs* deformable four-teat model). Fully covering the entire design space is not possible within the limited amount of time we had to our disposition. As such, we have proceeded by eliminating design options as early as possible in the process, and continued to refine the remaining (successful) options. A second dimension regarding the design space concerns the setting of the various *parameters* of the established algorithmic options. This dimension is discussed below.

VALIDATION: For a given algorithmic solution proposed for the AMD tracking problem, one needs to validate its results, *i.e.*, assess the quality of the tracking in terms of *e.g.* how well (accurately) teats are tracked and/or for how many frames of a given video. Separately, one needs to assess how the parameter setting of the proposed solution affects the quality of the tracking. In brief, we need an analysis of the space of input and output parameters of our algorithm to both validate and, where possible, improve its quality. This is, in itself, a complex problem which deserves extensive attention. As such, we treat this problem separately in the next Chapter.

OTHER APPROACHES: As already outlined in the related work, many approaches exist for object localization and tracking in computer vision. As such, and given that setting up any single such approach is relatively time-consuming, it was not possible to compare more extensively our proposal with such alternative approaches beyond the material already presented in this chapter. It is well possible that one of these existing computer vision methods does perform well for our use-case and type of data (however, this is very challenging, since as already explained many such methods have been designed for higher-resolution images). Separately, it is very likely that machine learning approaches, in particular deep learning, would yield good results for our tracking problem. However, such approaches require numerous labeled samples, which means in our case video frames with accurate associated 3D teat positions. As explained in this chapter, constructing such rich labeled data is very expensive, which makes the application of such methods impractical.

On the topic of designing an AMD tracker, several extension directions are possible atop of our solution presented here. Different teat detectors can be designed to find teats more accurately under extreme zoom-out conditions, *e.g.* based on a refinement of the 3D template matching proposed in Sec. 5.4.1.5. Secondly, using a more complex model including both teats and udder shape could render our tracking accuracy even higher in contexts of high occlusion. However, this will likely pose higher computational costs which have to be assessed in detail. Finally, deploying our algorithm implementation on embedded hardware, such as typically found in industrial robots, will very likely imply various algorithmic changes, so that our pipeline fits the limited computing power and available memory of such platforms. All such issues are part of future work.

6 VISUAL ANALYSIS FOR AMD TRACKER OPTIMIZATION

6.1 OVERVIEW

In Chapter 5, we have presented and end-to-end system for the detection of *trails* representing the 3D locations of cow teats, by using a computer vision pipeline based on the analysis of luminance and depth images acquired in real time with a Time-of-Flight (ToF) camera. As stated there, our system can achieve good performance – it tracks teats correctly in over 90% of the available frames.

However, the above statement, if left as such, would be hardly substantiated. First and foremost, we need a detailed quantitative evaluation of the tracker's *performance*. This, by itself, introduces a new question: How to quantify performance? Given our goal (accurately tracking the 3D positions of a cow's teats over time), performance can be defined as the accuracy with which each teat is tracked over a given video sequence. However, this definition introduces a new problem: To assess this accuracy, we need *ground truth*, *i.e.*, the actual 3D positions of a cow's teats over a given period of time. Unfortunately, such information is not available to us, nor is it easy to obtain, as regulations in the dairy industry forbid attaching markers to udders to collect such data by *e.g.* marker tracking. Hence, we need other ways to explore the output (3D teat trail-set) created by our tracker.

A second challenge is to *optimize* the proposed tracker. As outlined in Chapter 5, the trails produced by our tracker are affected by a number of design choices and parameter settings. As such, it is important to understand how the tracker output depends on these degrees of freedom. In particular, it is important to analyze the available parameter space so as to detect ways of improving the tracker's performance (on the one hand) and situations where performance, albeit low, cannot be further improved due to objective limitations of the input images (on the other hand).

To cover the above points, this chapter proposes the use of *visual analytics* to explore the large parameter space created by our tracker method's internal parameters, input data (images), and output data (3D teat trails). We proceed in Sec. 6.2 by presenting a visual analytics tool that supports this exploration process, allowing us to quantitatively evaluate our tracker's performance, detect and understand situations where performance is low, and also compare our performance with a brute-force accurate search method of the parameter space. Next, in Sec. 6.3, we extend this analysis by spotting suboptimal parameter configurations which can next be changed to improve the tracking performance. For this, we show how multidimensional projection techniques for visualizing multivariate datasets can be leveraged – an endeavor which, to our knowledge, has not yet been used in the task of optimizing shape-tracking systems. We conclude this chapter in Sec. 6.4 outlining our key contributions and findings.

6.2 QUANTITATIVE ASSESSMENT OF TRACKER PERFORMANCE

As already explained, analyzing the full tracking process is crucial to validate the robustness and correctness of our proposed solution – or, in other words, assess whether and/or how much of the trail-set data output by the tracker actually fit the image data captured by the ToF camera.

The video data we have as input for the tracker is unlabeled, *i.e.*, has no ground-truth for the correct teat positions. Labeling it would cost a huge effort (manually marking 3D teat positions in thousands of frames and/or 3D point clouds for several videos). As such, we need a different way to perform the validation.

For this, we propose a two-step procedure. First, we present in Sec. 6.2.1 a visual analytics tool that allows visualizing the tracked teat-set $\mathcal{M}$ (Sec. 5.4.2) and the different internal parameters of our tracker over time. We use this tool to get a first impression on the overall tracking performance, but also to pinpoint situations (combinations of input images and parameter values) where tracking has problems. Next, we use the tool to compare the performance of two different template detectors for finding teats in the depth image (Sec. 6.2.2). Finally, we extend our quantitative analysis of the tracking performance by comparing the results of our tracker to those produced by a completely different 3D tracking method based on brute-force accurate search (Sec. 6.2.3).

6.2.1 *Analysis Tool*

In this section, we give a description of the analysis tool we developed to assess the performance of our tracker. Its set-up follows the overview and details-on-demand design common for visual analytics tools [239], showing both overall tracking performance, but also finer-level details that explain this performance. The analysis tool is connected in a feedback loop with detection-and-tracking (Secs. 5.4.1, 5.4.2) so that the analyst can spot sub-optimal results in the overview, examine details to find their causes, adjust the responsible algorithm parameters, see the effects (*e.g.* improvements), and repeat the process until an optimal algorithm and parameter-set is found. As input, the analysis tool receives the entire data space that the tracker operates on, consisting of the luminance and depth video streams; derived images, match locations, error metrics, and system state (uninitialized, racking, initializing, or tracking lost); and the produced trail-set $\mathcal{M}(t)$. All these parameters vary in time, *i.e.*, take different values at each frame of the input video sequence. For an overview of all these parameters, we refer to Chapter 5.

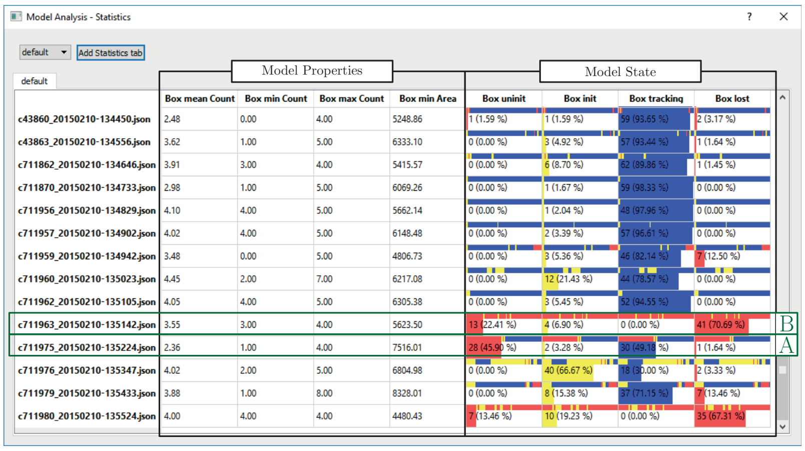
Our analysis tool consists of two main views, as follows. The first view gives a high-level overview of the tracking performance over multiple input sequences and/or parameter settings, as shown in Figure 6.1. The second view gives a more detailed overview of the performance for one selected result, like the example shown in Figure 6.2. The two-view design follows the well-known Shneiderman principle of overview-and-details [239]. We describe these two views next.

Overview: The high-level overview of tracking results are presented in the form of a table with a row for each dataset to compare. As explained, such a dataset can be either a different video, or the same video tracked with different tracker parameters. The columns of the table are configurable by the analyst and show high-level information (derived) from the analyzed datasets. This way, the high-level overview effectively allows comparing several datasets (rows) from the viewpoint of several metrics (columns). The column information can be split into two categories (as also shown in Figure 6.1) – *model state* and *model properties* – as follows:

MODEL STATE The model state view gives an overview of the self-reported performance of the tracker using its internal states of uninitialized, initializing, tracking, and tracking lost (see Sec. 5.4.2). This can be used to identify (parts of) tracking results where the system functions in a suboptimal way. We provide visual cues to help the analyst with this task in two ways. First, we indicate the percentage of frames that have a given state by coloring the same amount of the view's background to allow easier finding of the situation one is interested in, be it good or sub-optimal tracking, based on this visual cue alone. Secondly, we provide an overview of the tracking state over time as a small inset for each state indicator. As an example, we can see that the result marked *A* in Figure 6.1 has problems with initializing the tracking even when we look at the cell that indicates when tracking is successful, since we see a large red part at the beginning of the overview inset.

MODEL PROPERTIES The model properties view shows selected statistics computed over the results such as the minimum, maximum or mean value of a property over the entire duration of tracking. These values can for example be used to formulate a first hypothesis on what is the cause of sub-optimal tracking. For instance, the result marked *B* in Figure 6.1 shows bad performance with only a few frames where the model managed to initialize, but the next frames did not allow to continue tracking from this initialization. The model properties for this result show that there were enough teats at all times to be able to keep tracking – indeed, the minimum number of teats (Fig. 6.1, column *Box min Count*) is three. Hence, tracking was lost for the dataset *B* due to some other problems than not finding enough teats in the image. For such cases, the overview tells us that we need to study in detail other parameters to understand the problem.

Further study of the results in the overview is supported by allowing the analyst to sort them based on any column. For instance, if one wants to focus on problematic results, these can be found by ordering the table ascendingly on succesful tracking percentage, which puts problematic sequences (having lowest number of succesful tracking frames) at top. After a hypothesis about the cause of (sub-)optimal performance for a given tracking result has been formulated, a detailed study of this result can be started by opening the finer-level detailed view for this result.



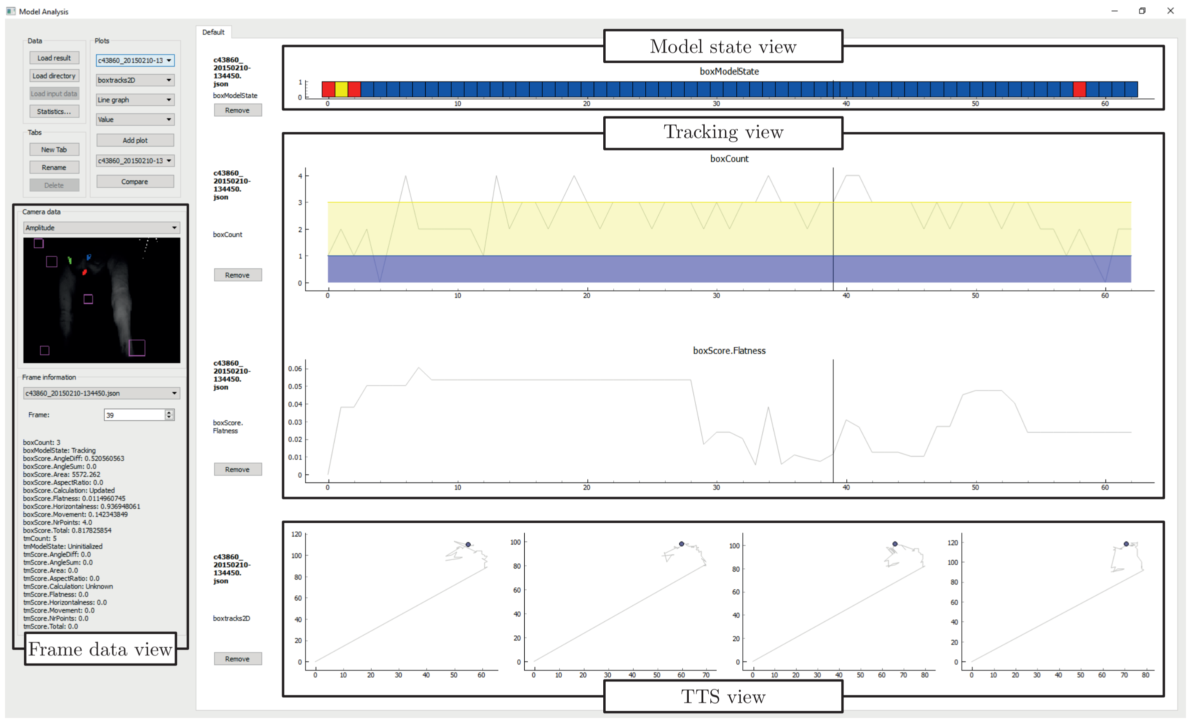
	Model Properties				Model State				
	Box mean Count	Box min Count	Box max Count	Box min Area	Box uninit	Box init	Box tracking	Box lost	
c43860_20150210-134450.json	2.48	0.00	4.00	5248.86	1 (1.59 %)	1 (1.59 %)	59 (93.65 %)	2 (3.17 %)	
c43863_20150210-134556.json	3.62	1.00	5.00	6333.10	0 (0.00 %)	3 (4.92 %)	57 (93.44 %)	1 (1.64 %)	
c711862_20150210-134646.json	3.91	3.00	4.00	5415.57	0 (0.00 %)	6 (8.70 %)	62 (89.86 %)	1 (1.45 %)	
c711870_20150210-134733.json	2.98	1.00	5.00	6069.26	0 (0.00 %)	1 (1.67 %)	59 (98.33 %)	0 (0.00 %)	
c711956_20150210-134829.json	4.10	4.00	5.00	5662.14	0 (0.00 %)	1 (2.04 %)	48 (97.96 %)	0 (0.00 %)	
c711957_20150210-134902.json	4.02	4.00	5.00	6148.48	0 (0.00 %)	2 (3.39 %)	57 (96.61 %)	0 (0.00 %)	
c711959_20150210-134942.json	3.48	0.00	5.00	4806.73	0 (0.00 %)	3 (5.36 %)	46 (82.14 %)	7 (12.50 %)	
c711960_20150210-135023.json	4.45	2.00	7.00	6217.08	0 (0.00 %)	12 (21.43 %)	44 (78.57 %)	0 (0.00 %)	
c711962_20150210-135105.json	4.05	4.00	5.00	6305.38	0 (0.00 %)	3 (5.45 %)	52 (94.55 %)	0 (0.00 %)	
c711963_20150210-135142.json	3.55	3.00	4.00	5623.50	13 (22.41 %)	4 (6.90 %)	0 (0.00 %)	41 (70.69 %)	B
c711975_20150210-135224.json	2.36	1.00	4.00	7516.01	28 (45.90 %)	2 (3.28 %)	30 (49.18 %)	1 (1.64 %)	A
c711976_20150210-135347.json	4.02	2.00	5.00	6804.98	0 (0.00 %)	40 (66.67 %)	18 (30.00 %)	2 (3.33 %)	
c711979_20150210-135433.json	3.88	1.00	8.00	8328.01	0 (0.00 %)	8 (15.38 %)	37 (71.15 %)	7 (13.46 %)	
c711980_20150210-135524.json	4.00	4.00	4.00	4480.43	7 (13.46 %)	10 (19.23 %)	0 (0.00 %)	35 (67.31 %)	

Figure 6.1: High-level overview of tracking results.

Detail view: The finer-level view of our analysis tool consists of several linked views as shown in Figure 6.2. These introduce two additional levels of detail, *i.e.*, the evolution of model properties over time, and detail information about a selected frame. To achieve this, we use four views, as follows:

MODEL STATE The *model state* view shows a timeline overview of the TTS model state (uninitialized, initializing, tracking, or tracking lost). States are shown by categorically color-coded bars, where blue means the model is succesfully tracking, yellow means the model has been (re-)initializated, and red means tracking is lost, respectively. This gives an easy-to-follow global overview of the entire tracking process and allows quickly spotting frames whose state changes from neighbor frames, *e.g.*, frames where tracking fails and which occur in a sequence of correctly tracked frames. After spotting such frames, one can use the views described below to find causes of the respective state-change.

TRACKING VIEW The *tracking view* refines the overview information from the model state view by showing graphs of all model parameters as functions of time. Correlating values of these signals with



```
Frame information
c43860_20150210-134450.json
Frame: 39

boxCount: 3
boxModelState: Tracking
boxScore.AngleDiff: 0.520560563
boxScore.AngleSum: 0.0
boxScore.Area: 5572.262
boxScore.AspectRatio: 0.0
boxScore.Calculation: Updated
boxScore.Flatness: 0.0114960745
boxScore.Horizontalness: 0.936948061
boxScore.Movement: 0.142343849
boxScore.NrPoints: 4.0
boxScore.Total: 0.817825854
tmCount: 5
tmModelState: Uninitialized
tmScore.AngleDiff: 0.0
tmScore.AngleSum: 0.0
tmScore.Area: 0.0
tmScore.AspectRatio: 0.0
tmScore.Calculation: Unknown
tmScore.Flatness: 0.0
tmScore.Horizontalness: 0.0
tmScore.Movement: 0.0
tmScore.NrPoints: 0.0
tmScore.Total: 0.0
```

Figure 6.2: Visual analysis tool for our teat detection-and-tracking system, having three overview views (model state, tracking view, and TTS view) and one detail view (frame data). All views are linked by interaction.

state values (or state changes) in the model view allows tracing the cause of the respective states one step back, *i.e.*, to the components of the error metrics E_{geom} or E_{motion} (subsection 5.4.2). For some model properties, extra visual cues are provided to help interpret the plot, such as the shaded regions in the tracking view showing the number of detected teats in Figure 6.2 that indicate the minimum amount of teats needed for initialization or tracking to succeed. Another possibility is to show the expected value for a parameter (such as the area between the teats) in the corresponding plot. This provides the analyst a visual indication of how close the actual parameter value is to the expected value.

TTS VIEW The *TTS view* shows the trajectories (trails) of the four tracked teats over the entire analyzed video, as 2D camera-view projections. We can also show 3D world-space trails of the tracked teats (see Figure 6.3, TTS view, bottom row). From actual tool usage, we found that the 2D trail projections are easier to interpret, so these are used as default in this view. Given the assumed smooth motion of both the tracked shape (cow) and camera (robot), the teat trajectories should be smooth curves. Also, these curves should have a relatively similar overall shape, given the geometric constraint that limits the relative motion of teats from each other (subsubsection 5.4.2.4). Hence, spotting large line-segment jumps in the TTS view allows us to find time-ranges when tracking performed incorrectly.

FRAME DATA The *frame data* view shows the raw luminance, depth, and point-cloud data acquired from the ToF camera for the frame selected in the other views, as well as numerical statistics on this frame (number of matches and values of the model metrics). These 'details on demand' allow refining the insight obtained from the overviews.

All views are linked by interactive selection – clicking on a time-instant or position in the overviews shows details of the selected frame in the frame data view. Similarly, the selected frame is also indicated in the other views. This is done for the model state and tracking view using a vertical line to indicate the selected frame. For the TTS view, the corresponding teat locations at the selected time-moment are indicated as dots along the plotted trails.

We next present an actual use-case that demonstrates how the detailed views an be used to spot and understand tracking problems. Figure 6.3 shows the detailed views with information loaded from a given tracking run. Three separate insights are discussed below.

Tracking lost due to robot steering: In the model-state view, we immediately see a suspiciously large amount of red (tracking lost) frames. At first sight, this suggests that our tracking is not working optimally. Let us focus on the largest red block, marked A in Figure 6.3. We see that this block correlates to a zero value for the $E_{flatness}$ metric (Equation 5.3), which is plotted in the tracking view. This tells us that tracking is lost because this metric had a too low value, which in turn caused E_{geom} to exceed the allowed threshold ϵ. Showing other model parameters in the tracking view allows back-tracing the cause of a large $E_{flatness}$ error to earlier data in the pipeline, such as the number and locations of found matches. Using this procedure for this dataset, we found out that, for the time-range of block A, the cause was that there were no correct matches found in the input image, due to the robot drifting out of the udder area. As we expect tracking to be lost in such cases, this does not flag a problem of our tracker, but a problem of the robot's steering.

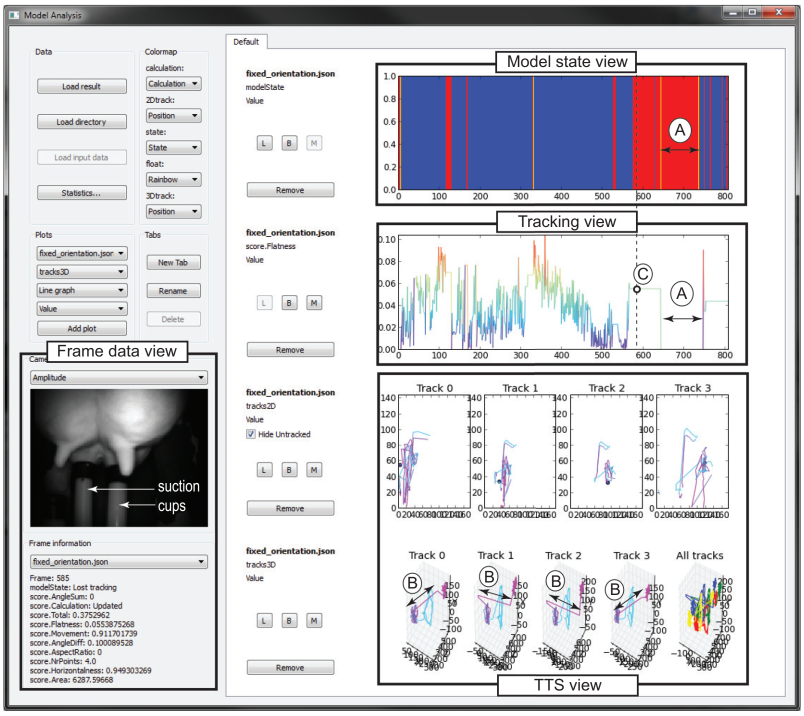
Undesired tracking-lost events: Using the TTS view, we can find areas where tracking is performing sub-optimally by looking for jumps in the teat tracks. Such a jump is marked B in Figure 6.3, and is visible for all four teats. Clicking on such a jump brings the data for the respective time moment(s) in focus in the other views. The current time is shown in the tracking view by the dot marker labeled C. We now see that this moment corresponds to the beginning of the first large red block in the tracking view. Hence, we know that the jump is caused by a tracking-lost event. If, however, the jump corresponded to a tracking state (blue in the model state view), this would have shown severe tracking problems, as the tracking would have created jumps (not in line with our knowledge of the studied phenomenon) *and* would have marked these as valid tracked frames.

Unavoidable tracking-lost events: Finally, the frame data view can be used to see if there is a relation between the input and the performance of the model. For instance, the frame data in Figure 6.3 corresponds to the moment C discussed above. As visible in the amplitude (luminance) image, the two back teats are now connected to the suction cups of the milking robot. In such cases, tracking is expected to be lost, due to the robot being too close to the udder and thereby occluding the teats. Hence, we have explained that the tracking-lost event observed in the TTS and model-state views is expected and not due to a tracker problem.

6.2.2 *Influence of Template Choice on Performance*

So far, we have used our analysis tool to find sequences or parts of sequences for which the performance is sub-optimal and used the same tool to find a relation between this performance and the input sequence or model parameters.

However, as explained in Chapter 5, the design space of a tracker is larger than just its parameter set. One other important degree of freedom is the choice of the template detector being used, which, as we have seen, can be a 2D one (Sec. 5.4.1.3) or a 3D one (Sec. 5.4.1.5). The related question is: Which, of these two detectors, delivers better performance in general?



fixed_orientation.json
modelState
Value

fixed_orientation.json
score.Flatness
Value

fixed_orientation.json
tracks2D
Value

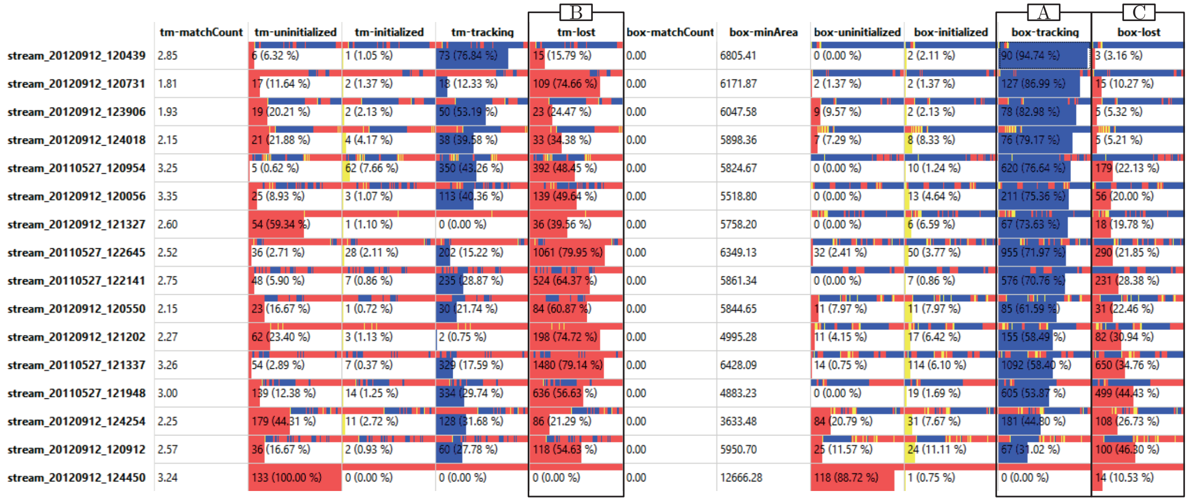
fixed_orientation.json
tracks3D
Value

Frame information

fixed_orientation.json

```
Frame: 585
modelState: Lost tracking
score.AngleSum: 0
score.Calculation: Updated
score.Total: 0.3752962
score.Flatness: 0.0553875268
score.Movement: 0.911701739
score.AngleDiff: 0.100089528
score.AspectRatio: 0
score.NrPoints: 4.0
score.Horizontalness: 0.949303269
score.Area: 6287.59668
```

Figure 6.3: Our visual analytics tool showing the finer-level details for a tracking result with sub-optimal performance.



	tm-matchCount	tm-uninitialized	tm-initialized	tm-tracking	tm-lost	box-matchCount	box-minArea	box-uninitialized	box-initialized	box-tracking	box-lost
stream_20120912_120439	2.85	6 (6.32 %)	1 (1.05 %)	73 (76.84 %)	15 (15.79 %)	0.00	6805.41	0 (0.00 %)	2 (2.11 %)	90 (94.74 %)	3 (3.16 %)
stream_20120912_120731	1.81	17 (11.64 %)	2 (1.37 %)	18 (12.33 %)	109 (74.66 %)	0.00	6171.87	2 (1.37 %)	2 (1.37 %)	127 (86.99 %)	15 (10.27 %)
stream_20120912_123906	1.93	19 (20.21 %)	2 (2.13 %)	50 (53.19 %)	23 (24.47 %)	0.00	6047.58	9 (9.57 %)	2 (2.13 %)	78 (82.98 %)	5 (5.32 %)
stream_20120912_124018	2.15	21 (21.88 %)	4 (4.17 %)	38 (39.58 %)	33 (34.38 %)	0.00	5898.36	7 (7.29 %)	8 (8.33 %)	76 (79.17 %)	5 (5.21 %)
stream_20110527_120954	3.25	5 (0.62 %)	62 (7.66 %)	350 (43.26 %)	392 (48.45 %)	0.00	5824.67	0 (0.00 %)	10 (1.24 %)	620 (76.64 %)	179 (22.13 %)
stream_20120912_120056	3.35	25 (8.93 %)	3 (1.07 %)	113 (40.36 %)	139 (49.64 %)	0.00	5518.80	0 (0.00 %)	13 (4.64 %)	211 (75.36 %)	56 (20.00 %)
stream_20120912_121327	2.60	54 (59.34 %)	1 (1.10 %)	0 (0.00 %)	36 (39.56 %)	0.00	5758.20	0 (0.00 %)	6 (6.59 %)	67 (73.63 %)	18 (19.78 %)
stream_20110527_122645	2.52	36 (2.71 %)	28 (2.11 %)	202 (15.22 %)	1061 (79.95 %)	0.00	6349.13	32 (2.41 %)	50 (3.77 %)	955 (71.97 %)	290 (21.85 %)
stream_20110527_122141	2.75	48 (5.90 %)	7 (0.86 %)	235 (28.87 %)	524 (64.37 %)	0.00	5861.34	0 (0.00 %)	7 (0.86 %)	576 (70.76 %)	231 (28.38 %)
stream_20120912_120550	2.15	23 (16.67 %)	1 (0.72 %)	30 (21.74 %)	84 (60.87 %)	0.00	5844.65	11 (7.97 %)	11 (7.97 %)	85 (61.59 %)	31 (22.46 %)
stream_20120912_121202	2.27	62 (23.40 %)	3 (1.13 %)	2 (0.75 %)	198 (74.72 %)	0.00	4995.28	11 (4.15 %)	17 (6.42 %)	155 (58.49 %)	82 (30.94 %)
stream_20110527_121337	3.26	54 (2.89 %)	7 (0.37 %)	329 (17.59 %)	1480 (79.14 %)	0.00	6428.09	14 (0.75 %)	114 (6.10 %)	1092 (58.40 %)	650 (34.76 %)
stream_20110527_121948	3.00	139 (12.38 %)	14 (1.25 %)	334 (29.74 %)	636 (56.63 %)	0.00	4883.23	0 (0.00 %)	19 (1.69 %)	605 (53.87 %)	499 (44.43 %)
stream_20120912_124254	2.25	179 (44.31 %)	11 (2.72 %)	128 (31.68 %)	86 (21.29 %)	0.00	3633.48	84 (20.79 %)	31 (7.67 %)	181 (44.80 %)	108 (26.73 %)
stream_20120912_120912	2.57	36 (16.67 %)	2 (0.93 %)	60 (27.78 %)	118 (54.63 %)	0.00	5950.70	25 (11.57 %)	24 (11.11 %)	67 (31.02 %)	100 (46.30 %)
stream_20120912_124450	3.24	133 (100.00 %)	0 (0.00 %)	0 (0.00 %)	0 (0.00 %)	0.00	12666.28	118 (88.72 %)	1 (0.75 %)	0 (0.00 %)	14 (10.53 %)

Figure 6.4: Using the high-level overview to compare the tracking performance of the 2D template method and the 3D template method, sorted by successful tracking performance of the 3D template based method.

To answer this, we constructed two trackers using the same set of parameters and algorithmic components, except the 2D *vs* 3D template detectors. Next we ran the trackers on a set of 16 recorded videos. The high-level overview of the results is shown in Figure 6.4. We chose to sort the results in descending order of tracking performance of the 3D tracker (column *A*, Fig. 6.4). The shape formed by the decreasing blue bars in this column shows that the 3D template tracker reaches very good performance for almost all of the videos, and completely fails tracking only for the bottom-most video. For the same sequence of videos, the 2D tracker always has a lower tracking performance.

Comparing columns *B* and *C* and the small insets with model the state overview in Figure 6.4, we can see that, when using the 3D template, detection is much more robust. When using the 2D template detection method, there are more frames for which tracking is lost, even if this total count is just a few frames. For the 3D template based detection, we see a much lower rate for this state of the detector.

We also examined the performance of tracking using 2D *vs* 3D templates at a finer level, using the various detailed views presented in Sec. 6.2.1. The initial observation that the performance of the 3D tracker is higher was confirmed. We noticed a few scenarios where the cause of this difference can be easily explained. For example, when the camera is pointed more upwards (towards the udder), the difference in depth between the teats and the surrounding background is decreased. Indeed, with a horizontal-pointing camera, the background consists of the cow hind limbs or barn walls, which are quite far away from the teats; with a more vertically-pointing camera, the background is the udder itself, which is very close to the teats. This decrease in contrast causes a strong decrease in performance for the 2D template based method which heavily relies on this contrast as it works in the gradient domain (see subsubsection 5.4.1.3). For such types of videos, the 3D template based detection works well, as it does not rely on contrast with respect to a background.

Summarizing the above, we found that the 3D template-based method is more robust for camera orientation, and performs better than the 2D template-based method. Additionally, the 3D method is easier to implement and has fewer parameters to control. Taken together, this makes it the candidate of choice in the tracker's design space.

6.2.3 *Ground Truth Comparison*

As outlined so far, using the 3D template detector, we developed a tracker that shows successful tracking in about 90% of the frames of all videos we tested it with. While a high tracking-state percentage is a necessary condition for a good tracker, it is not a sufficient one. That is, we need also to show that the tracked structures are indeed the teats, and not other spurious objects.

However, as also already explained, showing that our tracker tracks the correct objects requires ground truth data, which we do not have and cannot obtain. As such, we opt for a different solution: Given a video sequence that records the time-dependent positions $\tilde{\Omega}_i$ of an udder shape $\tilde{\Omega}$, we extract a highly-accurate sequence of 3D udder shapes Ω_i^{gt} that are very close to $\tilde{\Omega}_i$, by using a fine-grained (but expensive) parameter-space search. As such, we argue that Ω_i^{gt} is very close to the actual ground truth $\tilde{\Omega}_i$. Hence, we can measure the tracking accuracy of the results of our tracker by comparing them with the 'proxy' ground truth Ω_i^{gt}. We next describe the construction of the proxy ground truth (Sec. 6.2.3.1) and its comparison with our tracker results (Sec. 6.2.3.2).

6.2.3.1 *Constructing a proxy ground truth*

The key idea of computing our proxy ground truth sequence Ω_i^{gt} is to fit a high-resolution synthetic 3D udder model to the actual point cloud in each video frame i, subject to rigid transformations. Given the level of detail and the specific shape of this 3D model, a good fit (to the point cloud) cannot occur unless the model's teats match well the corresponding points in the point cloud. As such, we can use the teat positions in the fitted sequence Ω_i^{gt} as proxies for the actual teat positions in the video.

To construct the sequence Ω_i^{gt}, we proceed as follows (see also Figure 6.5). First, we select a video sequence in which the udder is very well visible in at least one frame – that is, has a depth image with no occlusions, as few self-occlusions as possible, no teat deformations, low noise levels, and a good distance to the camera. We next export the corresponding point cloud for this sequence, and manually crop the udder area from it. Next, we use a reconstruction method to extract a meshed surface M_Ω from this point cloud – in our concrete tests, the Poisson method [135] gave the best results. Surface reconstruction was discussed in Sec. 5.4.1.1. As noted there, a *fully automatic* surface reconstruction from the type of point clouds we have was not possible. However, this is not an issue in our case, where we can both manually crop the udder from spurious surrounding structures, and fine-tune the reconstruction method's parameters. The resulting mesh model M_Ω is shown in Figure 6.6.

Next, for each frame i of a video sequence, we construct the proxy ground truth Ω_i^{gt} by minimizing the difference

$$C(\tilde{\Omega}_i, \Omega_i^{gt}) = \frac{\sum_{\mathbf{x} \in P(\Omega_i^{gt})} (D_{\tilde{\Omega}_i}(\mathbf{x}) - D_{\Omega_i^{gt}}(\mathbf{x}))^2}{1 + |P(\Omega_i^{gt})|}. \tag{6.1}$$

Here, D_Ω indicates the pixel depth map of a shape Ω as seen from a given viewpoint; $P(\Omega)$ indicates all pixels covered by the screen-space projection, or rendering, of a shape Ω; and $|\cdot|$ denotes the number of pixels in a rendered image. Given the above, $D_{\Omega_i^{gt}}$ indicates the depth map of the proxy ground-truth shape Ω_i^{gt}, and $D_{\tilde{\Omega}_i}$ is the actual depth image taken by the camera

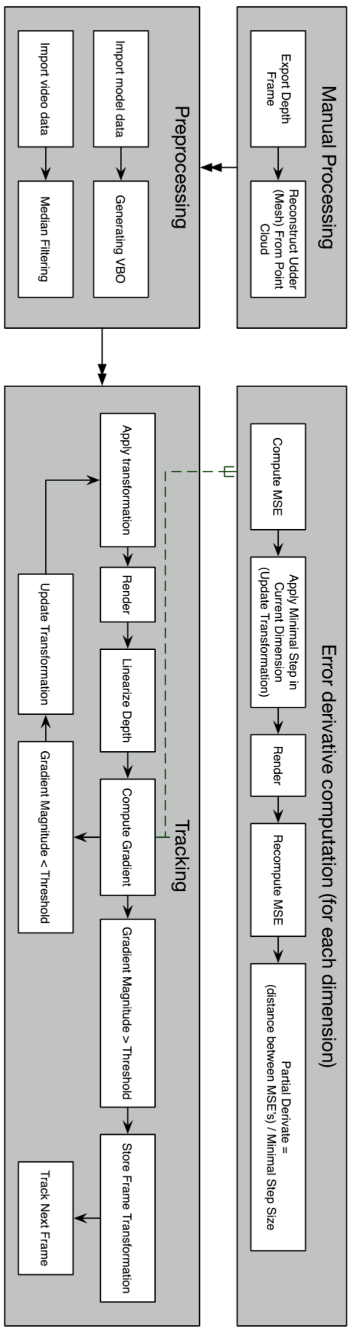

Manual Processing
Export Depth Frame
Reconstruct Udder (Mesh) From Point Cloud
Preprocessing
Import model data
Generating VBO
Import video data
Median Filtering
Error derivative computation (for each dimension)
Compute MSE
Apply Minimal Step in Current Dimension (Update Transformation)
Render
Recompute MSE
Partial Derivate = (distance between MSE's) / Minimal Step Size
Tracking
Apply transformation
Render
Linearize Depth
Compute Gradient
Gradient Magnitude > Threshold
Store Frame Transformation
Update Transformation
Gradient Magnitude < Threshold
Track Next Frame

Figure 6.5: Pipeline of the computation of the proxy ground truth (see Sec. 6.2.3.1).

which looks at the physical udder $\tilde{\Omega}_i$. Equation 6.1 thus computes the mean squared error (MSE), or difference, between the depth maps of the actual and synthetic udders, normalized by the size of (amount of pixels covered by) the synthetic shape Ω_i^{gt}. The difference C is zero when the synthetic shape Ω_i^{gt} perfectly matches the depth profile of the actual shape $\tilde{\Omega}_i$.

To minimize C, we parameterize the synthetic shape as

$$\Omega_i^{gt} = T(M_\Omega, \omega_i^j). \tag{6.2}$$

Here, M_Ω is, as explained, the high-accuracy mesh model of the udder; T represents a rigid transformation (translation, rotation, scaling); and and ω_i^j are the parameters of such transformations, *e.g.*, angle and axis of rotation for a rotation-transformation. Hence, Ω_i^{gt} is a rigidly transformed udder so as to best match the depth profile from the ToF camera in the current frame i.

The computation of the transformation parameters ω_i^j is performed using gradient descent. Gradient descent requires a starting value for these parameters. For this, we use their values ω_{i-1}^j from the previous frame. This works quite well, given the slow motion of the cow and the high frame-rate of the ToF camera, which together determine that the udder moves relatively little between consecutive frames. For the first frame $i = 0$, we initialize the parameters ω_0^j manually, by interactively changing them and visually monitoring the actual rendered image Ω_0^{gt} until it appears very close to the rendered image of $\tilde{\Omega}_0$. After this, gradient descent can be used to get an accurate fit, just as for the other frames.

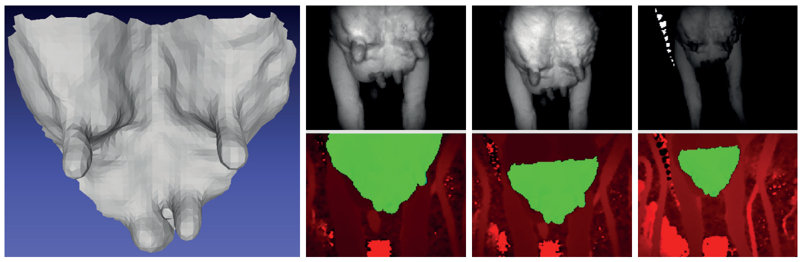


Figure 6.6: The 3D model fitting method in action, showing the udder model M_Ω and three different frames where the fitted model (in green) tracks the udder's depth map (in red).

A few implementation details follow. The transformations T (Eqn 6.2) are implemented as OpenGL scaling, rotation, and translations on the vertices of the mesh M_Ω. To obtain the depth map $D_{\Omega_i^{gt}}$, we render the transformed mesh Ω_i^{gt} in an offscreen OpenGL buffer, read the corresponding Z-buffer, and apply linearization to it to correct for the non-linear effects of typical fixed-precision Z-buffer schemes. The OpenGL camera parameters are set so as to emulate the actual ToF camera. The step sizes used by gradient descent are set to low values, which yields a slow convergence speed but high accuracy. Finally, the cost C (Eqn. 6.1) is computed using pixel-wise operations on the two depth images.

Using this set-up, we are able to track a cow udder for multiple input sequences with high precision (see Figure 6.6 for an example), providing us with the ground truth data we need to verify the performance of our tracking solutions described in Chapter 5. However, this method *cannot* be used as a tracker itself. Indeed, the method requires manual initialization in the first frame, and is also quite slow and computing-resoures hungry. For instance, when using only translation in the deformation function T (Eqn. 6.2), and a relatively low-resolution mesh M_Ω of 5000 vertices, fitting a single frame takes about 0.5 seconds on a modern PC. However, for our ground-truth computation, such limitations are not a problem.

6.2.3.2 *Tracker* vs *ground truth comparison*

Having the proxy ground-truth sequence Ω_i^{gt}, we now find the proxy ground-truth for the four teats by simply extracting the 3D trails (trajectories) of four vertices of the mesh M_Ω that we select, manually, by clicking on the visible terminations of the teats in this model. Since the mesh's topology does not change with the deformation T (Eqn. 6.2), these vertices correspond to the teat tips also in the deformed mesh $T(M_\Omega)$.

Next, we pair these four ground-truth trails with the actual trails of the tracked teat-set (TTS) $\mathcal{M}_i$ delivered by our tracker (see Sec. 5.4.2). This is easy to do, since for both the synthetic model M_Ω and the TTS we know the order in which teats come (left-to-right, front-to-back).

Finally, having these paired trails, we compute and display the 3D distances, over time, between the corresponding point pairs. Figure 6.7 shows such a plot, computed for the 3D template-based tracker, which, as discussed in Sec. 6.2.2, is our preferred design. Several observations can be made from this plot.

First, we see that the distances never become zero. It is important to understand that this cannot happen, since the brute-force (gradient) tracker and the 3D template-based one track different points which are close, but not identical to, the teats' tips. As such, we should interpret the graphs with care.

Secondly, we see that the four graphs, corresponding to the four teats, evolve relatively 'in sync'. In other words, the difference between the ground truth and tracker evolves similarly over time, for all teats. This can be seen as indiret evidence to the robustness of the tracker.

Thirdly, we see that the ratio of the largest to the smallest error (at any point in time) is roughly 3 to 1. Since the smallest error is caused by the aforementioned differences in definition of what is the tip of a teat, which is about the diameter of a teat (see Sec. 5.4.1.5), then the largest error is about 3 times this diameter. For the envisaged application (automatic milking devices), this is an acceptable error.

We can get further insight into the error behavior by analyzing separate regions of the graphs in Fig. 6.7. We find four such interesting regions showing different error behavior (denoted A, B, C, D in Fig. 6.7), as follows:

Region A: Here, we can see that our tracker agrees very well with the proxy grond truth – keeping, again, in mind the inevitable differences caused by different teat-tip definitions. The peak at the end of this region shows a single outlier frame with different behavior in the two sequences. However, even for this frame, the difference is well within the range of a teat's size.

Region B: We see here that the difference between the tracked teats and ground truth vary identically for all teats. Looking at the tracker's output (using the model state detail-view, see Sec. 6.2.1), we found that the tracker is in a 'tracking lost' state – thus, the tracked teat positions do not change in time, while the ground-truth does. While this is not a desirable situation, it does not indicate a *low accuracy* tracking. Indeed, when tracking is lost, one should simply ignore the tracker's output.

Region C: In this region, the errors are almost constant, *i.e.*, which suggests that the tracker is simply 'offset' away with a fixed distance from the proxy ground truth. Moreover, we see that two of the teats (green and purple graphs) are tracked with the same (good) accuracy, while the other two (blue and orange graphs) are tracked with different, and lower, accuracies. Examining the actual 3D udder model, we found that the good-acuray teats are the front ones, while the other two are the back ones (farthest from camera). After studying the individual frames from the actual input, the ground truth, and the tracker, we found the following explanations for the above. First, we note that the ground truth computation, which uses only rigid transformations, is of limited accuracy in cases where the udder deforms non-rigidly, which was the case in this temporal region. The 3D template-based tracker can deal with such situations quite well, given the deformable quad model used for the TTS (see Sec. 5.4.2). Moreover, visual inspection (of the limited time-extent of temporal region C) showed that the location of the back teats delivered by the 3D template tracker was actually more accurate than the proxy ground truth, as compared to the point-cloud delivered by the ToF camera. This could make one think that using the proxy ground truth computed by the method outlined in Sec. 6.2.3.1 is thus unreliable. We argue that this is not the case:

- When the tracker and ground truth match well, they should also match the actual data well. Indeed, since the two methods track using completely different methods (and largely different information types), the chance that they are both wrong *in a consistent way and at the same times* is, we believe, small;
- When the tracker and ground truth do not match well, we analyze both image sequences manually for the respetive period of time, and determine which one is wrong, and how much. While this process

requires some manual labor, it is far less intensive than manually analyzing the *entire* collection of videos to assess whether the tracker is accurate or not. In turn, this analysis can indicate moments when the proxy ground truth is incorrect (thus, we need to assess the accuracy of the tracker using purely visual information); and moments when the ground truth is correct but the tracker is not (thus, tell us valuable information about specific configurations that we need to refine the design for).

Region D: This last region shows a more rapidly-varying, high-frequency, error behavior. Looking at the actual video, we found that the teats here are quite close to the borders of the video image, a situation where the 3D template-based tracker is not handling robustly (see Sec. 5.4.1.5). Hence, the added value of ground-truth comparison is to quickly point us to such limitations of our tracker, which one can next focus on to alleviate.

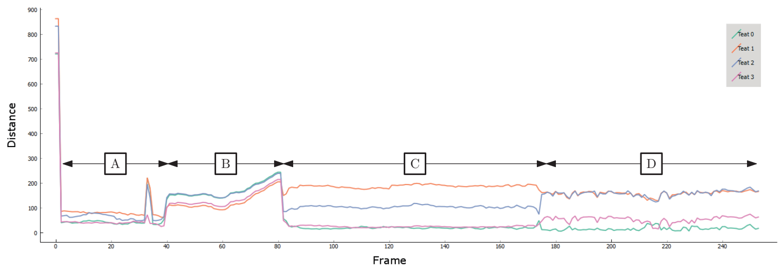


Figure 6.7: Distance between corresponding teats from the proxy ground-truth and the 3D template-based tracker, over time. Each graph, having a different color, indicates one teat.

6.3 PARAMETER SPACE ANALYSIS FOR TRACKER IMPROVEMENT

In Section 6.2, we have presented ways to quantitatively assess the performance of the tracker designs proposed in Chapter 5. Additional to performance estimation, this allows us to zoom in on specific sequences where tracking is suboptimal and trace back problem causes to the values of various parameters or input data configurations.

However, the visual analytics tool proposed for the above uses a *sequence-oriented* model: The data is organized as a set of sequences (of time-dependent parameter values and images). The top-down exploration proposed by the tool works by identifying one or a few sequences where tracking problems exist, followed by zooming on these specific problems one by one. This works well for a relatively small collection of short sequences. However, for a large collections of longer videos, this process is cumbersome, as it does not elicit *correlations* and *similarities* across different time moments and/or videos. In other words, the tool proposed in

Sec. 6.2 offers a deep, but narrow, view on the behavior of a tracker: One can *e.g.* select a small time-range of interest and then analyze how the various parameter values have caused a certain effect, *e.g.* a failed tracking. This is fine for examining isolated events, but does not help *generalizing* from such observations over a large collection. For instance, we can find a few events where tracking failed because the size of the 3D template is lower than a certain threshold τ. However, how do we know that *whenever* the template size is below τ, tracking will fail? Without knowing this, it does not make sense to start tuning the value of τ, as this may or may not fix the failed tracking.

To support this kind of generalization from individual observations (or tracker parameters), we propose next a different kind of visual analysis, which uses a *similarity-oriented* model: We model the entire set of time-dependent parameter values (from all tracking runs in a collection) as a set of high-dimensional observations $\mathbf{x}_i \in \mathcal{D}$, where $\mathcal{D}$ represents the union of parameter domains (typically, a subset of $\mathbb{R}^n$ for n such parameters), input video data, and output 3D trails. Each observation represents the entire state of the tracker at a given point in time and in a given sequence. Next, we visually explore this observation dataset using methods that emphasize similarities and correlations between observations, respectively parameter values, over the entire corpus of observations. Based on such findings, we can next draw generalizing conclusions over our tracker's behavior.

6.3.1 *Parameter Space Analysis with Multidimensional Projections*

Following the parallel between sequence-oriented and similarity-oriented noted above, we note a second parallel between the visual analysis introduced in Sec. 6.2 and the one we will propose next: The views presented in Sec. 6.2 provide a *variable-centric* exploration, i. e., they allow detailed examination of the evolution of a few variables over time. In contrast, the views we will introduce next take a *observation-centric* perspective: They abstract away (largely) from the chronological ordering of observations, and also from the individual variables, and focus on the similarity of states the system is in. This allows us to find correlations and similarities over arbitrary moments in time, and without choosing a subset of parameters of interest.

Key to this approach is the use of *multidimensional projections*, which have been introduced in Sec. 3.2.2.1. As explained there, projections scale very effectively to display large numbers of observations (hundreds of thousands), each having tens of dimensions or more. This is in contrast to other methods for visualizing high-dimensional data, such as tables (Sec. 3.2.1.1), parallel coordinate plots (Sec. 3.2.1.5), and scatterplots (Secs. 3.2.1.3, 3.2.1.4). Given that our datasets consist of tens of thousands of observations or more (basically, every frame in our tens of minute-length videos, taken at 24 frames per second, is an observation), and that observations have between 10 and 20 dimensions (all internal parameters of the tracker plus the 3D positions of the four detected teats), projections fit our data best as a visualization method. Equally importantly, they are observation-centric methods, which support tasks that relate observations among themselves, rather than examining individual parameters (Sec. 3.2.2).

By themselves, raw projections of high-dimensional data are of little use – they can show clusters of similar observations, but not why these are similar. As such, we use next a more advanced visual exploration tool which enhances projections with several interactive visual explanatory and exploratory mechanisms. The tool itself, called *featured*, has been developed for the goal of assessing (and improving) the performance of classification systems for static (time-independent) image data using machine learning [244]. Our use here is entirely novel, as we employ it to assess (and improve) the performance of tracking systems for time-dependent 3D trail-sets.

Analysis tool: We next present a brief description of the *featured* visual analytics system in [244], and its adaptions to our problem and data context. The tool offers several linked views, as follows:

OBSERVATION VIEW The observation view shows the raw observations that the tracker receives as input. As explained in Sec. 5.3, these consist of a set of time-dependent luminance images I_i, depth images D_i, and point clouds P_i. From these, we choose to display only the luminance images I_i, as these are the easiest to interpret visually, and thus match the goal of the observation view. Images I_i in this view are sorted on the index i, thus, chronologically, as they appear in a video sequence;

FEATURE VIEW The feature view shows all parameters that describe the state of the tracker, as described in Sec. 5.3[1]. Parameters are grouped into

- *input:* these are values computed from the input image. Besides the number of matches in the match-set M_i (Sec. 5.4.1.3), we add here various image characterization features known in the imaging literature, such as average intensity, average edge strength, and histogram of oriented gradients (HOG), computed over small image patches of about 16×16 pixels. The added value of these features will become apparent in the discussion below;
- *output:* these are values that the tracker computes, such as the positions of the four vertices of the tracked teat-set (TTS) $\mathcal{M}_i$ (Sec. 5.4.2);
- *model:* these are internal values of the tracker itself, such as the TTS area $A(\mathcal{M}_i)$ (Eqn. 5.2); the various energy components that

1 The terms feature, parameter, and dimension have the same meaning, and refer to independent measurements over observations. We use here the term feature so as to be in line with the terminology employed by the *featured* tool [244].

quantify the TTS quality (E_{motion} (Eqn. 5.1), E_{shape} (Eqn. 5.2), $E_{flatness}$ (Eqn. 5.3), E_{orient} (Eqn. 5.4), and E_{geom} (Eqn. 5.5)); and the tracker state (uninitialized, initializing, tracking, or tracking lost). Besides the tracker state, which is a categorical variable, all others are quantitative variables.

The feature view allows selecting several such features, along which we want to explore the observations, using the views described next.

OBSERVATION PROJECTION VIEW The observation projection view shows a 2D scatterplot of the projection of all observations using the features selected in the feature view. This allows seeing how the available observations (time moments in the currently analyzed video(s)) are similar from the perspective of the selected features. Observations can be colored in this view based on the value of a selected feature. For instance, in Fig. 6.8, observations are colored based on the categorical 'tracker state' attribute. To account for the different ranges of the input data (dimensions), these are normalized by standardization, as usual when using multidimensional projections. Different projection techniques are available to use, among which we note MDS, LAMP, and t-SNE. In the following images, we use t-SNE as an example.

FEATURE PROJECTION VIEW The feature projection view shows a 2D scatterplot where each point represents a feature of our data. In contrast to the observation projection view, where distance between points reflects the *Euclidean distance* (in the high-dimensional feature space) between observations, the feature projection view uses as distance the *correlation* between features, measured over all observations. This allows seeing which features are strongly correlated (or not).

GROUP VIEW This view allows one to group observations based on various criteria. For instance, we can group observations based on their similarity as shown in the observation projection view, *e.g.* by interactively selecting observations in that view which form an isolated cluster; or we can group observations based on the value of one feature, *e.g.*, we can select all time moments (frames) where the tracking was lost.

Analysis scenario: We next present how we can use the *featured* tool to analyze the high-dimensional information available in a tracking run to understand causes of tracking problems and proposed improvements to such problems.

We start by importing the tracking data into *featured* (Figure 6.8). Next, we are interested to analyze how observations are similar (or different) with respect to the tracker's state. As such, we select all features in the *model* category in the feature view, and obtain a projection (Figure 6.8, observation

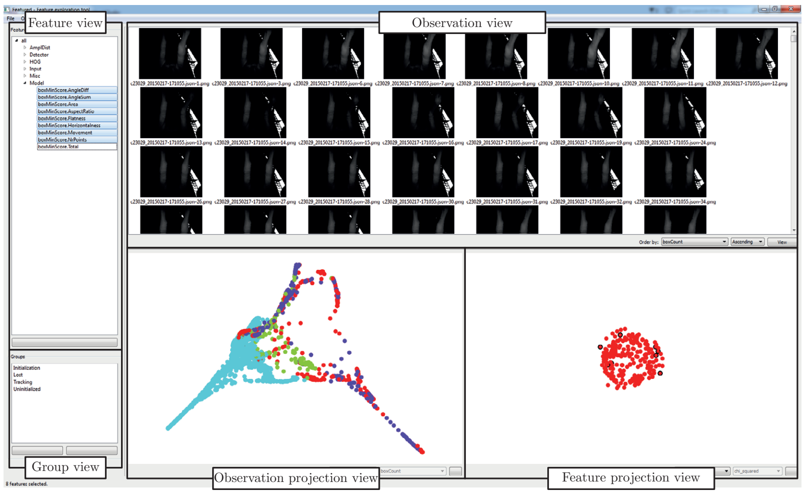


Figure 6.8: The *featured* analysis tool after loading in the data generated by teat tracking system.

projection view). We know, however, that the total geometric error E_{geom} is a linear combination of other three errors which are already included in the model feature-set (see Eqn. 5.5). As such, the similarity of observations in the observation projection view should not depend on using the additional feature E_{geom}. We verify this by removing E_{geom} from the set of features used to construct this projection and by noticing that the observation projection indeed does not change.

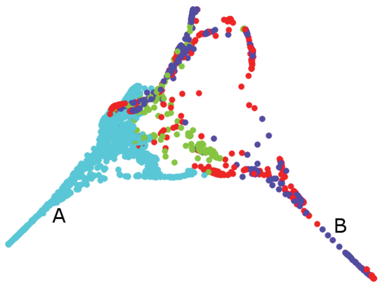


Figure 6.9: Projection of all tracking-model features computed by the tracking model showing the segregation of observations (frames) into a cluster of 'tracking' frames (A) and a cluster of 'uninitialized' and 'tracking lost' frames (B).

We are interested, at a high level, to understand how the tracker *operates*. If we look at the observation projection, we see two salient regions with a similar "pointy" shape which are clearly separated from the rest of the observations ((Figure 6.9), markers A and B). This is a first sign that the tracker's model parameters describe quite different configurations in state space – indeed, if all states were similar, from the perspective of the model parameters, the projection (which, as noted, employs Euclidean distances based on model parameters between observations) would show an unstructured ball-like point cloud, as well known from the dimensionality reduction literature.

Next, we are interested to understand what makes the two point clusters A and B so different. To do this, we color the points in the observation projection view by the value of the tracker state feature. We now see that region A is almost completely cyan, which corresponds to the state 'tracking'; and region B is a mix of red (state 'uninitialized') and purple (state 'tracking lost'). Green points correspond to the 'initializing' state which, unsurprisingly, appears to have parameter values located roughly in between the tracking and not-tracking states. Since the regions A and B are quite far apart in the projection, this means that their model parameters are also quite different. In other words, we found that tracking fails (uninitialized or lost) because of specific value-ranges of the computed model parameters, which are quite different from those for states where we can track. As such, we know that our current model parameters are a good indicator of the ability to track or not.

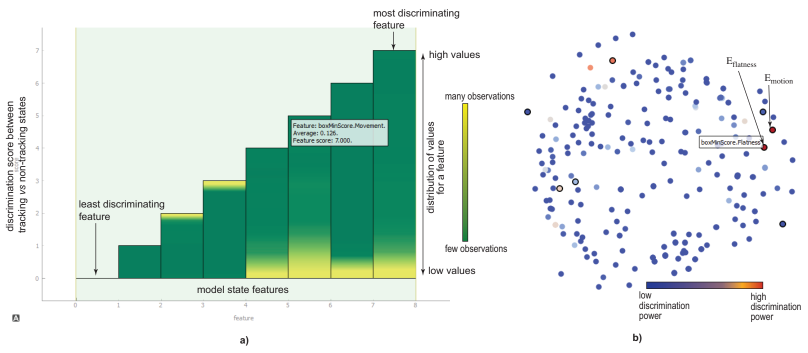


Figure 6.10: Importance of features when comparing the 'tracking' state with the other states of the tracker. (a) Feature scoring bar chart. (b) Feature projection view.

As we now know that the states where we can track are very different, parameter-wise, from those where we cannot, the next question is which specific features and/or feature values discriminate between the two types of states. To find this out, we select all observations labeled as 'tracking' and rank all model features by their ability to discriminate between these observations and the remaining ones. This can be done by using so-called *feature scoring* techniques. Specifically for our case, we use univariate techniques (one-way ANOVA), and so-called wrapper techniques, based on the utilization of a classification algorithm (randomized decision trees [95], recursive feature elimination (RFE) [96], and randomized logistic regression [176]).

According to all these feature scoring techniques (except ANOVA), the most discriminatory feature between tracking and non-tracking states is E_{motion} (Eqn. 5.1), with $E_{flatness}$ (Eqn. 5.3) being a close second. For ANOVA, the situation is very similar – $E_{flatness}$ scores highest, followed by E_{motion}.

Figure 6.10(left) shows the scores of eight features, computed with RFE. Each feature is represented as a bar whose height indicates the score. The two rightmost bars represent $E_{flatness}$ and E_{motion} respectively. Bars are colored to show the distribution of values of a respective feature using a green (few) to yellow (many) values. Thus, we see that both $E_{flatness}$ and E_{motion} have most values in their lowest respective ranges (yellow is at the bottom of these bars). Further coloring the observation projection by the values of these two features (in turn) shows that the low-value ranges match well the tracking state, while high-value ranges match the non-tracking states, respectively.

Summarizing the above findings, we have learned that

1. tracker parameters strongly correlate with states being separated into tracking *vs* non-tracking ones;

2. the two state types are most discriminated by the values of $E_{flatness}$ and E_{motion};

3. tracking states have low values of $E_{flatness}$ and E_{motion}, while non-tracking states have high values thereof;

4. state parameters vary continuously over the observation space (indeed, we do not see clearly-separated clusters in Fig. 6.9).

The above findings match well the ideas behind our tracker design in Chapter 5, which was based on the assumption that the TTS quadrilateral $\mathcal{M}_i$ is relatively flat *and* moves slowly between frames. As such, when these conditions occur, we can track well, as our above analysis shows. Conversely, when these conditions do not occur, we loose tracking. Now that we understand the strong correlation of tracking success with these parameter values, several improvements can be done. First, we can increase the maximum error allowed for these metrics (see Eqns. 5.1 and 5.5 and related text). This, indeed, allows tracking more deformed and/or faster-moving udder configurations, and is a simple but effective solution. Secondly, we could design better correspondence finding methods (see Sec. 5.4.2.2) that allow robust pairing of TTS configurations between consecutive frames when high motion amounts are found. We have not explored this direction, however.

The feature scoring analysis provides, next, other insightful points. Besides the above-mentioned two features which strongly discriminate between tracking and non-tracking states, we have, as described earlier in this section, several other model parameters. These do not have a clear correlation with the tracking *vs* non-tracking states. This does not mean, of course, that they do not have a role in the *design* of the tracker. However, they do not have critical *values* that correlate with the tracking success. As such, we know that putting effort into fine-tuning their computation and/or defining acceptable thresholds for them is not highly likely to optimize the tracker. While this does not make our tracker better, it saves effort that has a low chance to do so.

We now explore the feature-values' correlation with the tracking *vs* non-tracking states using the feature projection view (Fig. 6.10). Here, as explained, a dot is an entire feature (over all observations). We include here all available features from our system, thus including input, output, and model features. In total, this gives us over 200 features. Next, we color features by their discriminative power between tracking *vs* non-tracking states, using a blue (low power) to red (high power) colormap. The result shows several interesting insights. First, we confirm the earlier finding that the two most discriminative features are E_{motion} and $E_{flatness}$ – these correspond to the dark red dots in the figure. However, more interesting is the fact that none of the *input* and *output* features are strongly correlated with the tracking *vs* non-tracking dichotomy. Since we have a rich set of features describing both input and output, this tells us that, in principle, there is little in the input data (images) or in the computed teat trails that corresponds to situations where we can track *vs* cannot track. As such, a (cautious) inference is that the tracker's success is largely based on the way we interpret the flatness and motion-speed parameters. This supports our earlier observation that relaxing the thresholding criteria on these parameters can improve the tracking success. The feature projection view (Fig. 6.10) shows another interesting thing: As there are no clearly separated clusters in this scatterplot, it means that the features under study are largely uncorrelated (independent). This is a positive finding, as it tells us that we do not have redundancy (features that would measure the same property) in our tracker design.

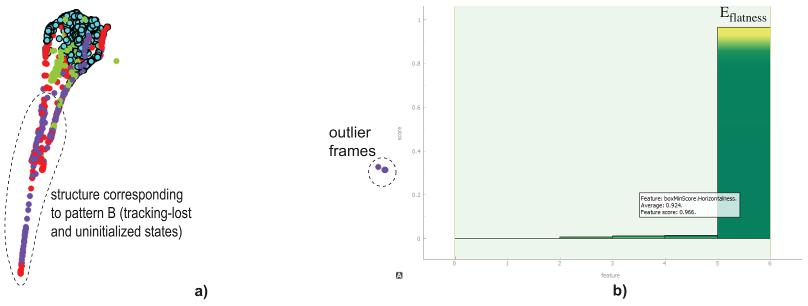


Figure 6.11: (a) Observation projection excluding feature E_{motion}. (b) Feature scoring (ANOVA) to discriminate outliers (see image (a)) *vs* rest.

Let us now further disentangle the impact of the two most important features that determine the success of tracking – $E_{flatness}$ and E_{motion}. As we have seen, E_{motion} is related to the speed of motion of the udder. This is relatively hard to constrain in reality. Moreover, it is expected that during a normal milking procedure, cows move little; this was indeed not the case for our testing sequences, but this was mainly due to the experimental set-up used for the ToF camera during testing, which caused the animals to swing more than normally expected. However, $E_{flatness}$ is much harder to constrain, as cow udders do often exhibit teats of quite different lengths (see *e.g.* Fig. 6.6 left).

To study the separate effect of $E_{flatness}$, we remove E_{motion} from the set of features selected in the feature view. As explained, this regenerates the observation projection to ignore this feature. Figure 6.11a shows the result, color-coded by state, just as in Fig. 6.9. The distinctive ‘tail’ pattern formed by red (‘uninitialized’) and purple (‘tracking lost’) states visible in Fig. 6.9 is also present here. However, the cyan ‘tail’ earlier present in Fig. 6.9, which contained ‘tracking’ frames, is now collapsed to a ball, and mixed with ‘initializing’ (green) frames. This is indeed logical: Non-tracking (‘uninitialized’ and ‘tracking lost’) states do not carry any motion information, since there is no previous successfully-tracked frame with respect to which motion of the TTS $\mathcal{M}_i$ can be computed (see Sec. 5.4.2.2). As such, the structure of these observations does not change if we ignore E_{motion}, which anyways has zero values over them. However, the collapse of ‘tracking’ observations into a ball structure when removing E_{motion} tells us something very interesting: The tail structure A in Fig. 6.9 was determined *mainly* by different values of this parameter. Since this structure is (almost) unidimensional (an elongated spike), this tells that the main factor that distinguishes between ‘tracking’ frames is the motion speed, E_{motion}. However, since all observations in structure A in Fig. 6.9 are cyan (‘tracking’), it means that tracking works reliably for a large range of E_{motion} values.

The observation projection in Fig. 6.11a also shows us a few isolated outlier observations (frames) located to the right of the main bulk of observations. Being purple, these indicate frames where tracking failed. If we select these outliers and use feature scoring to find out which features make them so very different from the remaining frames, we get the answer that the overwhelmingly dominating cause is the value of $E_{flatness}$, see Fig. 6.11b (scoring computed using ANOVA). To further understand why this happens, we click on the respective outliers and use the observation view (see Fig. 6.8 top-right) to display their corresponding images. We find that, in these images, there are very difficult to detect teats (even for a human observer), or even no teats at all. Hence, we find another added-value of the projection-based analysis of tracking data as opposed to the first visual analytics tool we proposed (Sec. 6.2.1): We can now quickly find outlier frames without having to examine *all* the tracker state parameters.

6.4 CONCLUSION

In this chapter, we have first introduced a visual analytics tool for the assessment of performance of the tracker designs proposed in Chapter 5 (Sec. 6.2). The tool provides a top-down, overview-and-details, parameter-centric, exploration of all data involved in the tracking process, enabling users to locate problematic configurations and sequences, and explain the tracking problems in terms of the values of the various parameters used in the tracker, or alternatively in terms of special configurations present in the input images. The key added value of this approach is to help users *narrow down* the focus of exploring for problems (and their causes) to a small set of parameters, parameter values, and input configurations.

Separately, we have introduced a way to assess a tracker's performance quantitatively at several levels of detail. First, we do this by offering an overview of the percentage of frames where tracking succeeds over an entire collection of videos, which allows one to get a quick statistical impression of the tracker's robustness on real-world data. Secondly, we proposed a way to generate proxy ground truth, using an expensive (but high accuracy and robustness) brute-force tracker. Comparing our actual tracker's results with this ground truth yields a finer-grained view of the accuracy and robustness of the former, and also points to corner cases which should be improved.

Next, we have proposed an extension of the parameter-centric visual analytics system with an observation-centric view (Sec. 6.3). This extension allows a more global, inter-sequence and intra-sequence, analysis of the recorded tracker data, to support finding correlations between parameter values and finding similarities of system states. Together, these extensions allow one to generalize from individual findings so as to gather more general evidence supporting the detection of general tracker problems, and directions to improve them. In particular, this helped us to find how the tracker success (tracking *vs* non tracking) critically depends on two main parameters (and their values), one of which we can fine-tune to improve tracking effectiveness; that tracking success is largely independent on the low-level image characteristics of the input video sequence; and that we do not have clearly redundant parameters in our design. All these findings help, implicitly and/or explicitly, reducing the effort of analyzing and/or improving our tracking system. To our knowledge, this is the first time that a visual analytics approach was used to understand the working, and assess the performance, of tracking systems.

Overall, we have tested over 100 real-life videos of varying lengths (tens of seconds to minutes), each acquired in actual stables in a production-process environment, that cover a wide range of cow udder anatomies, camera-to-subject distances, angles, lighting conditions, and motion paths. Using the above-mentioned accuracy analysis, we found that the performance of our tracker amounts to **over 90%** of the frames being successfully tracked. This clearly exceeds the documented performance of comparable systems [111, 159, 178, 229, 290]. Moreover, for the tracked frames, our tracker delivers position information which is within an acceptable margin of error for the envisaged application. Determining these results, and also finding the few remaining (10%) outlier situations where tracker performance was suboptimal, and the reasons thereof, would not have been efficiently possible without the usage of the visual analytics systems proposed in this chapter.

Many directions for further work exist based on the results presented here. First and foremost, the visual analytics approach for tracker performance assessment, understanding, and improvement can be further exploited to all these three ends. Our work has shown that important insights in all these directions an be easily obtained, but has surely not worked out these insights to their final conclusions. To do this, one would need to iterate the insight discovery, design refinement, and tracker performance re-evaluation pipeline, or the so-called 'analytics sensemaking loop', several times. This can be directly done using no more than the tooling and methodology proposed here. Secondly, promoting the various constants used in the tracker design, such as thresholds, window sizes, and weights (see Sec. 5.3) to *parameters* evolving over given ranges, and incorporating these in the parameter analysis proposed in this chapter, would complete the scope of tracker design analysis. This way, one could have a full picture of how *all* the elements involved in our design presented in Chapter 5 actually affect the quality of the tracking results, and further be able to select better values for them that lead to higher quality tracking.

7 BUNDLING FOR SIMPLIFIED VISUALIZATION OF TRAIL AND GRAPH DATA

In Chapters 5 and 6, we have shown how we can extract 3D trails of a complex moving shape (the teats of a deformable cow udder), respectively how we can use visual analytics to assess the accuracy of such trails. In this chapter, we turn our focus from the visual analysis of trails at the 'micro' scale to the 'macro' scale. We look here at significantly larger datasets having hundreds up to a million trails, which describe the motion of vehicles over large geospatial areas. As such, our visual analytics focus moves now from supporting the assessment of the performance and accuracy of the trail-extraction process to the scalable visual presentation and exploration of trails. To this end, we propose to use a *bundling* approach. We present a new bundling method for large trail datasets, and explain how computational efficiency and limited clutter can be achieved by several techniques[1].

7.1 INTRODUCTION

Very large graphs and networks have become pervasive in data-intensive applications such as traffic and network monitoring, software engineering, bioinformatics, and telecom applications. When the size of such datasets exceeds certain limits, understanding them becomes challenging. Edge bundling methods have become an important tool for this task: Given a large graph having a 2D spatial embedding of its nodes, bundling produces a simplified view of the graph structure by grouping spatially-close and semantically-related edges, so that edge-crossing clutter is reduced and the graph's main connectivity patterns get easier visible. Bundling was used for applications in vehicle traffic [115, 142, 225], program understanding [50, 218], multivariate data analysis [168], and medical visualization [27].

While many bundling methods have been proposed, several key challenges exist to their usability and usefulness:

Scalability: Recent techniques can bundle graphs of tens of thousands of edges in subsecond time [92, 116, 184]. While impressive, this speed is still insufficient for graphs such as Internet connectivity patterns, worldwide traffic flows, or call graphs of large software systems of millions of

1 The material in this chapter is based on the publications: M. van der Zwan, V. Codreanu, and A. Telea. CUBu: Universal real-time bundling for large graphs. *IEEE TVCG*, 22(12): 2250–2263, 2016; and M. van der Zwan, A. Telea, and T. Isenberg. Continuous navigation of nested abstraction levels. In M. Meyer and T. Weinkauf, editors, *Proc. EG/IEEE VGTC Conference on Visualization (EuroVis) – Short papers*, pages 13–17, 2012.

edges. Moreover, bundling time-dependent (dynamic) graphs, or changing bundling parameters during interactive data exploration, asks for bundling methods that are one to two orders of magnitude faster. Such methods do not exist yet.

Directions: Most bundling methods cannot separately bundle edges having different directions. This creates a high amount of *overdraw*, which precludes users from reasoning about the directions of edges in a given bundle [119, 173]. Current directional bundling methods, which aim to solve this issue, are however too slow to cope with interactive exploration of large graphs [206, 209, 231].

Level of detail: As bundling highlights the main connectivity patterns in a graph, users also need *level-of-detail* techniques able to emphasize the importance of a given bundle for the overall pattern. Various shading techniques have been used for this, *e.g.* colormapping and alpha blending [107, 119, 142] and shaded tubes [76, 267]. While detail shading can effectively provide level-of-detail cues, it cannot (yet) be computed in real time and it is relatively complex to implement.

Generality: A final challenge is the proliferation of bundling techniques. While each such technique may excel in specific ways *e.g.* speed, ease of use, or achieving specific constraints on the resulting layout, achieving *all* these goals with a *single* algorithm is still hard [310]. Hence, users face the dilemma of implementing and using a large set of algorithms, or settling with the (dis)advantages of a specific algorithm.

We propose a single bundling algorithm: CUBu (CUDA-based Universal Bundling) to address all above challenges[2]. We tackle scalability by a GPU bundling method that achieves average speed-ups of 50 up to 100 times *vs* the fastest existing general-graph bundling techniques [92, 116, 184]. Next, we propose two directional bundling extensions that are fast, robust, and easy to use. Thirdly, we show how to create multiscale bundled visualizations having the same quality as comparable methods [76, 267] and with a much simpler, and faster, implementation. Finally, we show how to create bundle styles proposed by widely different methods [53, 76, 107, 142] with our single method. As CUBu achieves all the above, we dub it an 'universal' bundling algorithm. We illustrate CUBu's speed and quality by several applications on large real-world graphs.

The structure of this chapter is as follows. Section 7.2 covers related work. Section 7.3 describes our general bundling algorithm. Section 7.4 presents applications of CUBu in several domains. Section 7.5 discusses our method. Section 7.6 discusses the usage of trail bundling in the context of simplified (abstract) representations of flow fields. Section 7.8 concludes the chapter.

2 The acronym relates also to the spelling of the syntagm 'the cube' in Romanian, a tribute to the geometric interests and background of the last author of [315].

7.2 RELATED WORK

We overview existing bundling methods based on the four feature, or requirements, classes listed in Sec. 7.1:

Scalability: Early bundling methods for compound graphs, *e.g.* HEB[107], achieve a high scalability, by exploiting explicit hierarchical information present in the input graph. Methods for bundling general graphs (without hierarchical information) evolved from slow approaches, such as force-directed edge bundling (FDEB [108]), to advanced schemes to detect edge proximity and thus achieve faster bundling, such as control meshes (GBEB [53]), Voronoi diagrams (WR [142, 143]), medial axes (SBEB[76]), and radial kernel splatting (KDEEB [119], 3DHEB [184]). The MINGLE method uses multilevel edge clustering to accelerate the bundling process [92]. Scalability is critical for *e.g.* streaming (time-dependent) graphs: For the well-known *US airlines* dataset (900 edges), FDEB [108] achieves 19 seconds/frame on a 1.7GHz PC (see [108], Sec. 4.2); StreamEB achieves 6 seconds/frame on similar hardware [187]; KDEEB reaches 0.17 seconds/frame [119]; and 3DHEB yields 0.4 seconds/frame [184]. Such speeds are yet insufficient for real-time bundling of large static graphs or large dynamic graphs having hundreds of thousands of edges or more, such as the *wiki* or *amazon* graphs discussed in [92].

Directions: A bundle contains several edges placed close to or atop of each other. Bundles separate high-density edge groups by large white space areas, and thereby help perceiving the overall graph structure. Yet, bundling creates an undesired *overdraw* issue: We cannot display edge-specific attributes for (all) edges in a bundle, since these share the same screen space. This, in turn, makes it harder to reason about a bundle's semantics. The standard solution to this issue is to aggregate attributes at overlapping edges *e.g.* using averaging done by alpha blending [107, 108, 142, 267]. While this works well for quantitative scalar attributes, it yields wrong results for categorical attributes. Edge *directions* are one such example. To address this, one can separate same-direction edges into different bundles and next directionally color-code edges to show directions. Such *directional* bundling methods include divided edge bundling (DEB), which extends FDEB to include edge-direction compatibility [231]; attribute-driven edge bundling (ADEB), which extends KDEEB by a flow map encoding local edge-compatibility metrics [206]; and 3DHEB, which bins edges per-direction-interval and uses KDEEB to bundle each bin separately [184]. Yet, all these methods are much slower than undirected edge bundling, making them unsuitable for interactive large-graph exploration.

Level of detail: Graph splatting first proposed to depict a graph as a continuous image-space density field, computed by convolving the graph drawing by a Gaussian filter whose size controls the level-of-detail [282].

Accuracy issues on computing such dof the node ensity fields are discussed in [144, 225]. This image-space idea was extended in the LaGO tool by aggregating edges connecting local node-density maxima [312]. Recent bundling methods highlight a graph's main structure (dense bundles *vs* isolated edges) by alpha blending [107, 108, 142]. Shaded cushions [283], adapted to create crisp curved-and-shaded tubes along bundles [76, 267], better separate different pattern scales in a graph than splatting or alpha blending, and also separate crossing bundles better than alpha blending or straight-line edge aggregation. Yet, creating shaded bundle tubes is very complex, involving operations such as distance transforms and medial axes [76, 267], which are far from real-time performance.

Generality: Globally, we see three major styles in existing bundling methods: (1) *smooth* bundles, having few inflection points between end-nodes, are created by FDEB, KDEEB, and WR, and help easy visual following of large bundles; (2) strongly ramified bundles, showing a hourglass-like pattern between end-nodes, are created by HEB, GBEB, and SBEB, and help an easy detection of connection branching points; and (3) straight-line bundles, showing a highly simplified graph connectivity pattern, are created by LaGO, and help an easy recognition of end-to-end node connections. All these drawing styles have their merits (and limitations). The problem is that obtaining each such drawing style entails changing the bundling method being used. This is undesirable in terms of application-design simplicity and also forces users to learn parameter settings of many such methods.

7.3 PROPOSED METHOD

CUBu's input is a graph drawing $G = (V, E)$ with vertices $V = \{\mathbf{v}_i \in \mathbb{R}^2\}$ and edges $E = \{\mathbf{e}_i \subset \mathbb{R}^2\}$. Edges $\mathbf{e}_i$ can be straight lines or 2D curves, which covers bundling of both straight-line graphs [53, 108, 142] and spatial trajectories [115, 118, 119, 225]. Edges are modeled as uniformly-sampled polylines with control points, or *sites* $\{\mathbf{x}_j \in E\}$, and can be either directed or undirected. Our algorithm's first phase creates a bundling $B \subset \mathbb{R}^2$ of G, based on user-specified preferences: bundling density, directional separation, and desired bundle shape (Sec. 7.3.1). For this, we propose a set of techniques which lead to massive performance improvements of CUBu as compared to all existing bundling methods. In the second phase, we render B using suitable shading, transparency, and color-mapping to emphasize various structures of interest on different spatial scales (Sec. 7.3.2). These two phases are detailed next.

7.3.1 *Bundling algorithm*

To achieve our goal of real-time bundling of large graphs, we efficiently use parallel computing architectures such as NVidia's CUDA or OpenCL. After studying the wide family of general-graph bundling methods, we found that

KDEEB, one of the fastest bundling methods, is the most suitable to such parallelization. Yet, a careful study of KDEEB reveals several performance and accuracy issues. We next describe these issues, and how CUBu corrects them.

KDEEB follows the mean shift principle [46]: The edge sites $\mathbf{x}_j$ are convolved with a 1D parabolic (Epanechnikov) radial kernel K of radius R, to obtain an edge density map $\rho : \mathbb{R}^2 \to \mathbb{R}^+$ as

$$\rho(\mathbf{x} \in \mathbb{R}^2) = \int_{\mathbf{y} \in E} K\left(\frac{\|\mathbf{x} - \mathbf{y}\|}{R}\right) d\mathbf{y}. \tag{7.1}$$

Next, the sites $\mathbf{x}_j$ are advected upwards in the normalized gradient of ρ with a distance R, or in other words

$$\mathbf{x}_j^{new} = \mathbf{x}_j + R\frac{\nabla\rho}{\|\nabla\rho\|}. \tag{7.2}$$

Edges are next resampled to get an uniform and dense spatial distribution of the sites $\mathbf{x}_j$ over G, needed for a good kernel density estimation. Finally, a 1D Laplacian filter is applied on the edges to remove small-scale jitters created by the finite-step advection (Eqn. 7.2). The above three steps are repeated $p_N = 10$ times, while R is decreased from an initial user-specified value p_R by a small fraction at each step, so that, when all iterations are completed, the final R equals one. We next discuss our changes to all these steps, and their reasons to be.

Density map: KDEEB computes ρ using OpenGL by splatting the radial kernels K, encoded as 2D floating-point textures, at the site locations $\mathbf{x}_j$, and accumulating the result ρ in a floating-point image buffer. While simple to implement, this takes about 40% of the total bundling time for the graphs discussed in [116]. For large graphs of over 1M sites, we measured that splatting reaches over 60% of the total bundling time. This is due to the fact that splatting uses a *scattering* model: Data from sites $\mathbf{x}_j$ is scattered to neighbor pixels within the kernel radius R, so parallelization is severely limited by the many concurrent image-writes. We address this by a *gathering* strategy, following earlier results that indicate that this is more computationally efficient than scattering [198]: We compute $\rho(\mathbf{y})$ for each pixel $\mathbf{y}$ in the image ρ by summing up the contributions of all sites $\mathbf{x}_j$ closer to $\mathbf{y}$ than R. Additionally, we split the 2D convolution (Eqn. 7.1) into two 1D passes, one over the rows and one over the columns of the image ρ, and treat several blocks of rows, respectively columns, in parallel via CUDA kernel invocations.

For gathering to work, we need to know, for each pixel $\mathbf{y}$, all sites $\mathbf{x}_j$ for which $\|\mathbf{y} - \mathbf{x}_j\| \leq R$. Typical solutions for this use spatial search structures, *e.g.* kd-trees. While such techniques exist on CUDA [37, 93], they all need costly reinitializations after each advection step that moves the sites (Eqn. 7.2). We propose a faster solution: We first create a per-pixel site-density buffer $C : \mathbb{R}^2 \to \mathbb{N}$, where $C(\mathbf{x})$ gives the number of sites in E that

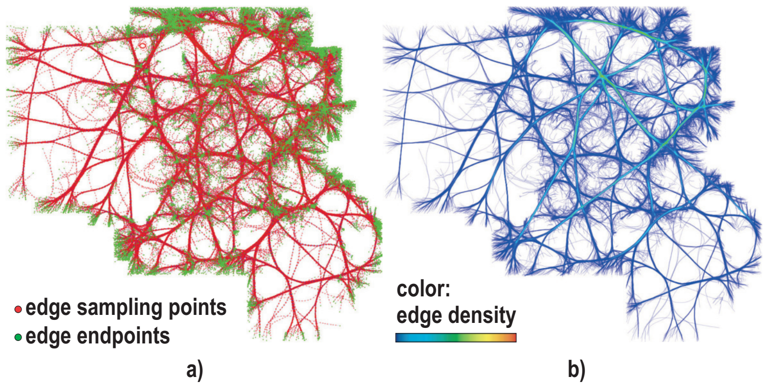


Figure 7.1: a) Bundled graph with nodes (green) and edge sampling points (green). b) Density map for the same graph.

fall inside pixel $\mathbf{x}$, by rendering all sites into the image C, using `atomicAdd` operations to take care of concurrent writes. Next, we compute $\rho(\mathbf{y})$ as

$$\rho(\mathbf{y}) = \sum_{\mathbf{x} \in T(\mathbf{y})} K(\|\mathbf{x} - \mathbf{y}\|) C(\mathbf{x}), \tag{7.3}$$

with $T(\mathbf{y})$ being a disk of radius R centered at $\mathbf{y}$. In contrast to scattering, where speed is bounded by the integral of ρ (Eqn. 7.1) over $\mathbb{R}^2$, the speed of our gathering is bounded by the much lower number of concurrent writes occurring when computing C, *i.e.*, the probability that two or more sites fall over the same pixel, *i.e.* the number of edge intersections in the graph drawing. To further decrease this probability, rather than sampling edges uniformly with a step of Δ units (as in KDEEB), we use a sampling step of $\Delta + \frac{\Delta}{10}\delta$, where δ is a random real number uniformly distributed in $[-1, 1]$, computed by CUDA's *cuRAND* library. This decreases the chance that closely-spaced edges, appearing in latter bundling iterations, have clusters of closely-spaced sites, separated by gaps of size Δ. A good by-product of our random sampling is that sites are more evenly distributed in image space, thereby leading to a better estimation of the density gradient $\nabla\rho$ used for advection.

Advection, resampling, and smoothing: KDEEB advection (Eqn. 7.2) strictly follows the mean-shift idea, *i.e.* moves sites upwards in the density gradient. Yet, since $\operatorname{div} \nabla\rho$ is practically never zero for our bundles (see analysis in [76]), advection increases the local edge-sampling density in negative-divergence areas, and decreases density in positive-divergence areas. To ensure a nearly constant sampling density (important for density estimation [116]), KDEEB resamples edges after *each* advection iteration. We measured this resampling cost on a wide family of graphs, and found it to be about 30% of the total bundling time, in line with [116]. Let us analyze what happens when advecting a site: The site's shift $\mathbf{x}_j^{new} - \mathbf{x}_j$ (Eqn. 7.2) can be decomposed into a drift along the tangent

$$\tau(\mathbf{x_j}) = \frac{\mathbf{x}_{j+1} - \mathbf{x}_j}{\|\mathbf{x}_{j+1} - \mathbf{x}_j\|} \tag{7.4}$$

of the graph edge sampled by $\mathbf{x}_j$ and a motion along the normal $\mathbf{n}_j$ to the edge at $\mathbf{x}_j$. Tangent drift $((\mathbf{x}_j^{new} - \mathbf{x}_j) \cdot \tau(\mathbf{x}_j))\tau(\mathbf{x}_j)$ causes sampling-density variations. We cancel this drift by advecting sites along $\mathbf{n}_j$, *i.e.*, replace $R\nabla\rho/\|\nabla\rho\|$ in Eqn. 7.2 by its projection along $\mathbf{n}_j$. This change, which has negligible computational cost, yields a much more uniform sampling density. We can now resample edges every three or four advection iterations, instead of every iteration, as in KDEEB. We also do one Laplacian smoothing iteration every 3..4 iterations rather that after each iteration (as in KDEEB), since our advection is controlled not just by the (imprecise) density gradient, but also by the shape of the edges themselves (due to normal projection). Our constrained advection gives a performance boost of about 20-30% as compared to KDEEB.

Bundle shape control: Different bundling techniques create different bundle *styles*, in terms of their shape, curvature, and thickness: HEB creates typical 'hourglass' shapes by its B-spline control polygons that capture the underlying hierarchy tree. HEB shapes have been found effective for tasks involving finding high-level connections between node groups [50, 270]. HEB bundles also constrain edge directions close to their node endpoints, helping one to visually match edges with node glyphs. FDEB and related methods, *e.g.* KDEEB, create bundles with less inflection points and smoother curvature variation than HEB, which are easier to follow visually [97, 108, 116, 187]. SBEB, GBEB, and WR create highly ramified bundles, which are good in showing splitting/merging of paths between node groups. While not typically seen as bundling methods, several techniques route spatially close edges along constrained paths, yielding highly simplified graph drawings [142, 312]. Summarizing, different bundle styles support different analysis tasks and/or user preferences.

Changing between bundle styles is hard. Small changes can be done by parameter tuning, *e.g.* the amount of Laplacian smoothing or edge relaxation [76, 108]. More complex style changes, *e.g.* creating HEB-style bundles with FDEB or a schematic bundled layout with KDEEB, are hard or not even possible without changing the bundling method. We achieve all such styles by changing a few parameters in CUBu.

HEB style: To create HEB bundles, we modulate site advection by an edge-profile function, *i.e.*, replace Eqn. 7.2 by

$$\mathbf{x}_j^{new} = \mathbf{x}_j + R\lambda(t(\mathbf{x}_j))\frac{\nabla\rho}{\|\nabla\rho\|}. \tag{7.5}$$

Here, $\lambda : [0,1] \rightarrow [0,1]$ controls the amount of motion of each site $\mathbf{x}_j$ as a function of the parametric arc-length position $t \in [0,1]$ of $\mathbf{x}_j$ along its

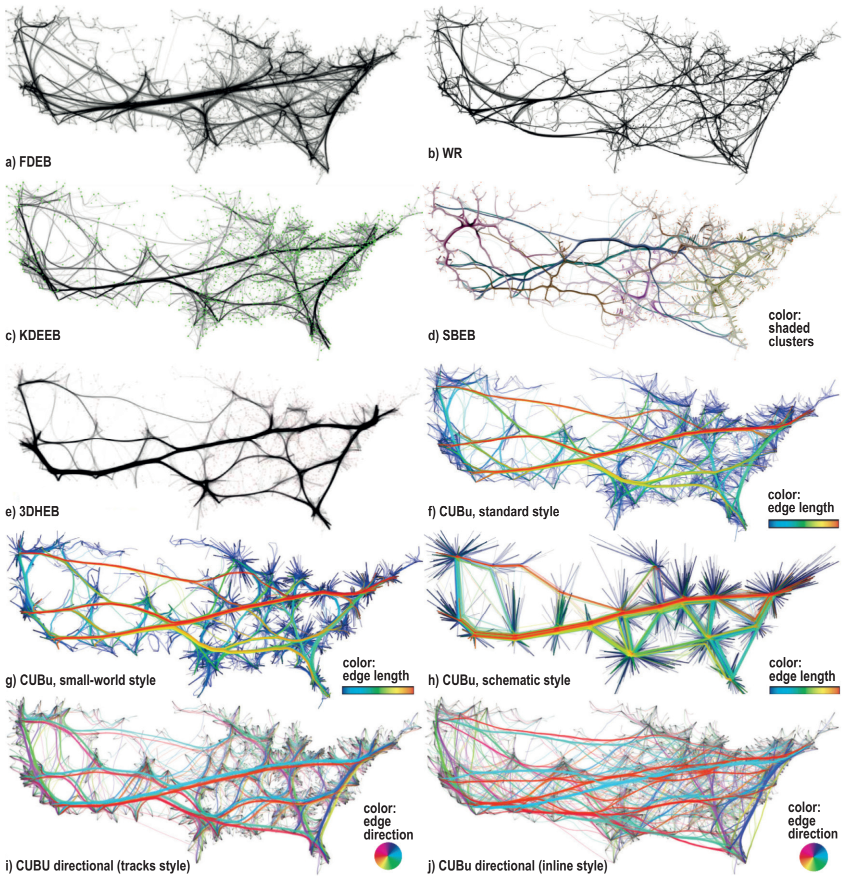


Figure 7.2: Bundling styles for *migrations* graph. (a-e) Existing algorithms. (f-j) Styles produced by our single CUBu method.

edge. Using a hourglass-like function $\lambda_{HEB}(t) = ((1-2|t-0.5|)^3)^4$, having two symmetric inflections at its endpoints $t = 0$ and $t = 1$ and a plateau of value 1 in the middle, creates HEB-style bundles, by gradually limiting the advection of sites close to bundle endpoints. Using a function $\lambda = 1$ produces the classical smooth fan-out common to general-graph bundling algorithms [76, 108, 116, 142].

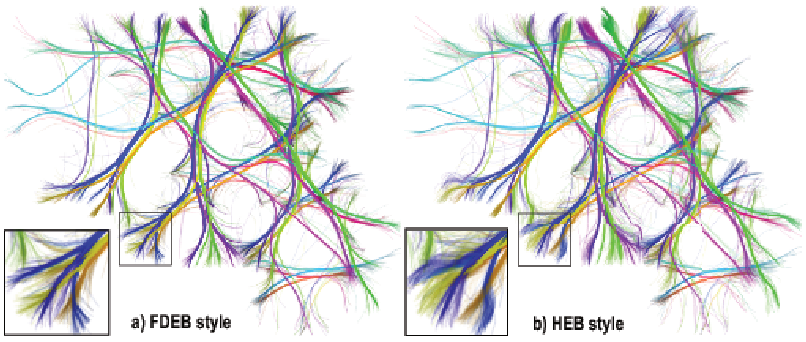


Figure 7.3: FDEB *vs* HEB styles for *airlines* graph, using 'parallel tracks' directional bundling.

Small-world style: When exploring small-world graphs, one wants to see how a compact and strongly-related node group is connected (or not) to other node groups. Several methods do this by pre-clustering nodes based on connectivity and distance, and next drawing straight-line connections between nodes and their cluster centers, followed by drawing connections between cluster centers themselves [12, 98, 99, 312]. This yields a typical 'linked star' pattern akin to the one in a bubble graph layout [26], where stars show node clusters and links between star centers show higher-level connections between node clusters. We obtain the same effect by edge bundling, without needing to explicitly cluster the input graph, as follows. Let $\mathbf{e}$ be an edge with node endpoints $\mathbf{v}_i$ and $\mathbf{v}_j$. We apply first do mean-shift clustering on all nodes $\mathbf{v} \in V$. This shifts each node $\mathbf{v}$ to a location $\mathbf{v}^c$ close to the center of its local neighborhood. Next, we insert the points $\mathbf{v}_i^c$ and $\mathbf{v}_j^c$ as the second, respectively one-but-last, sites on the sampled edge $(\mathbf{v}_i, \mathbf{v}_j)$. Finally, we apply our CUBu bundling method on the resulting graph drawing.

Figure 7.2g, shows the effect of this technique. Several star structures appear, showing groups of related nodes. Bundles now link the centers of these stars, showing the node-group to node-group main connectivity patterns more explicitly than via standard bundling (*e.g.* FDEB or KDEEB). More whitespace is left between the bundles, as edges will first automatically agglomerate by going to the node-group centers $\mathbf{v}_i^c$ and $\mathbf{v}_j^c$ prior to bundling proper. This further helps better visual separation of bundles. The kernel radius p_R used for mean shift clustering of edge endpoints gives the desired size of node neighborhoods: Large p_R values create fewer and larger node clusters, *i.e.* show the coarse-level small-world graph structure. Small p_R values create more and smaller node clusters, *i.e.* show the fine-level small-world structure. The lower bound of $p_R \leq 1$ pixels yields the standard KDEEB bundling.

Schematic style: Schematic bundled-graph drawing uses simple edge shapes and, at the same time, groups spatially close edges into bundles. Examples are orthogonal layouts used for software diagrams [60] and metro map layouts [188, 254]. CUBu can generate a particular type of schematic drawings, in which (a) spatially close edges are bundled and (b) bundles have shapes given by simple polylines consisting of a few segments. For instance, Fig. 7.2h can be seen as the schematic simplification of the bundlings in Figs. 7.2f or 7.2g. To do this, we simply edge resampling after each advection iteration. As outlined in Sec. 7.3.1, this makes advection create a highly non-uniform sampling-point density consisting of a few spatially-separated dense point clusters. Edges consist of segments linking such clusters, thus have the desired coarse polyline structure.

Directional bundling: Drawing graphs with edges separated by direction is of recognized importance. Only a few bundling methods can do this: DEB [231] adapts FDEB's edge-compatibility function [108] to add directional similarity atop of spatial closeness. However, this method is quite slow. Similarly, ADEB [206] and 3DHEB [184] extend KDEEB's edge-density function to include a flow field capturing edge directions (ADEB) and respectively compute H edge-density maps, uniformly sampled over the 2π edge orientation range. FDEB and 3DHEB are 5 to 10 times slower than KDEEB, although implemented on the GPU. Other fast bundling techniques, *e.g.* MINGLE, are hard to adapt to use directional compatibility. We describe next two directional bundling techniques that can be easily added to CUBu to produce similar results to DEB, 3DHEB, and ADEB, while keeping scalability.

Parallel tracks: Given a graph with edges $\mathbf{e}_i$ and sites $\mathbf{x}_j$, bundled by an undirected bundling method (*e.g.* FDEB, SBEB, KDEEB, WR, or any similar method), we move each site $\mathbf{x}_j$ with a small distance $\epsilon\lambda(t(\mathbf{x}_j))$ in the direction $\tau(\mathbf{x}_j) \times \mathbf{d}$, where $\tau(\mathbf{x}_j)$ is the normalized tangent vector to $\mathbf{e}_i$ at $\mathbf{x}_j$ (Eqn. 7.4), $\mathbf{d} = (0, 0, 1)$ is the vector normal to the 2D layout plane, and t and λ are the edge arc-length parameterization and edge-profile functions λ used earlier for bundle shape control. This effectively 'splits' each bundle into two parallel railway-like 'tracks' separated by a maximal distance 2ϵ, so that all edges in a bundle that go in the same direction stay in one such track. As Figs. 7.3 and 7.2i show, this creates a uniform and regular separation of bundles into two thinner, parallel-running, bundles which can be next easily color-coded to show edge directions. The value ϵ is controlled by the user, typically set to about 10 pixels for good results. Parallel tracks can be applied to any bundling method with negligible cost.

Inline directional bundling: Our second directional bundling technique takes place during bundling itself rather than in postprocessing. For this, we modify Eqn. 7.3 to compute, for each edge site $\mathbf{y}$, a density gradient $\rho(\mathbf{y})$ that accounts only for sites $\mathbf{x}$ within a radius R from $\mathbf{y}$ which have an edge-tangent vector $\tau(\mathbf{x})$ (Eqn. 7.4) that is compatible with the tangent $\tau(\mathbf{y})$. In detail, we replace Eqn. 7.3 by

$$\rho(\mathbf{y}) = \sum_{\mathbf{x}\in T(\mathbf{y})} K(\|\mathbf{x}-\mathbf{y}\|)C(\mathbf{x})\kappa(\mathbf{x},\mathbf{y}), \tag{7.6}$$

where $\kappa(\mathbf{x},\mathbf{y}) = \tau(\mathbf{x})\cdot\tau(\mathbf{y}) \in [-1, 1]$. This bundles close same-direction edge fragments as usual, but forces close opposite-direction edge fragments to repel each other. In the above, we use for $\tau(\cdot)$ the tangent directions of the *original* (unbundled) edges, as we want to estimate directional compatibility based on the input, and not on the bundled, graph.

Compared to parallel tracks (Fig. 7.2i), inline directional bundling (Fig. 7.2j) creates larger separations between edges having different directions, and an overall more natural effect, quite similar to DEB. Bundle shapes can now adapt more freely, as they are only constrained by directionally compatible edges.

7.3.2 *Visualization enhancements for bundled graphs*

We next propose three visual additions that help exploring the information conveyed by bundled graphs.

Color and opacity: Existing bundling techniques use color to encode geometric edge properties, *e.g.* direction or length; or edge attributes, *e.g.* time, height, or speed when bundling trail datasets [119, 142]. Alpha blending typically shows edge density ρ, *i.e.* bundle importance, as opacity [107]. Yet, tuning alpha blending is not easy: Too high values make the drawing cluttered in high edge-density areas [73]; too low values make outliers, like sparse bundles and isolated edges, hard to see.

We propose an enhanced color-and-opacity mapping scheme that alleviates the above issues. Each edge site $\mathbf{x}$ has an $HSVA$ (hue, saturation, value, alpha) attribute. We allow setting H and S based on any edge attribute. Examples below include both local and global attributes, such as the edge direction at $\mathbf{x}$ and the edge length respectively. V and A are set using a parabolic cushion profile function c, where

$$V(\mathbf{x}) = {}^{l}/_{l_{max}} + (1 - {}^{l}/_{l_{max}})c(\mathbf{x}), \tag{7.7}$$

$$A(\mathbf{x}) = \alpha \cdot (1 - {}^{l}/_{l_{max}} + {}^{l}/_{l_{max}} c(\mathbf{x})), \tag{7.8}$$

$$c(\mathbf{x}) = \sqrt{1 - 2\cdot|t(\mathbf{x}) - {}^{1}/_{2}|}. \tag{7.9}$$

Here, l is the length of the current edge; l_{max} is the length of the longest edge in E; t is the edge arc-length parameterization explained in Sec. 7.3.1; and $\alpha \in [0, 1]$ is a parameter that controls the drawing's overall opacity. Short edges get constant opacity A but a parabolic luminance V profile, whose gradient makes them appear more salient in the image, which helps spotting isolated outliers. Long edges get a flat luminance V and parabolic opacity profile. This way, their end fragments become less opaque, and thus make their connections to their end-nodes be more visible. Their flat luminance de-emphasizes them, as opposed to short edges, since their length is a strong enough visual cue to make them visible. Figures 7.2f-j show how this works: Compared to Figs. 7.2a-e, which use classical alpha blending, we now see many more detail edges that connect to isolated nodes.

Multiscale shading: Edges in bundled drawings can be hard to follow end-to-end due to crossings [173, 267]. This can be helped by a shading effect that makes bundles appear like 3D overlaid tubes rather than flat 2D shapes. The shading gradient (high across a bundle, low along it) makes the visual separation between crossing bundles easier. This can be done by explicitly computing separated bundles, by clustering edges belonging to the same bundle, and rendering generalized shaded cushions atop such bundles [283]. Such shading involves complicated and costly operations: edge clustering, distance transforms, and medial axes [76, 267]. For graphs of tens of thousands of edges or more, doing this at interactive rates is not possible.

We propose a new fast way to compute shaded bundles from a bundled graph drawing. Consider the density map ρ (Eqn. 7.1) as a height plot surface $z = \rho(\mathbf{x})$. At each pixel $\mathbf{x}$, we estimate the normal $\mathbf{n}(\mathbf{x})$ to this surface as the normalized 3D vector $\left(\frac{d\rho}{dx}, \frac{d\rho}{dy}, -1\right)$, with derivatives of ρ computed by central differences. We next use $\mathbf{n}(\mathbf{x}_i)$ to shade each edge site $\mathbf{x}_i$ by the classical Phong lighting model, like [142]. However, the method in [142] has a key problem: The density ρ can vary hugely over its domain: Along dense and tight bundles, ρ derivatives have high values, yielding almost horizontal normals, thus no illumination (if we use a vertical light vector); along sparse bundles of just a few edges, ρ derivatives get very low, yielding an almost vertical normal $\mathbf{n}$, thus maximal illumination. This is opposite to our goal to de-emphasize dense bundles and emphasize sparse ones. We handle this by a different shading model: For each pixel $\mathbf{x}$, we compute the maximal value $\rho_{max}(\mathbf{x})$ of ρ over a disk of radius r centered at $\mathbf{x}$. Next, we use $\rho(\mathbf{x})/\rho_{max}(\mathbf{x})$ as height in the above lighting computations. This locally normalizes ρ so that dense bundles appear as shallower bumps and sparser ones as taller ones, respectively. The parameter r controls the smoothing amount in a multiscale way: Larger values make bundles have larger highlights and less sharply-defined borders; smaller values create sharper highlights and bundle borders. Computing bundle shading is easily implemented in CUDA by a single pass over all pixels $\mathbf{x}$ of the density image ρ. Figure 7.4 shows our results. As visible, dense (important) bundles stand out easily. in terms of color and shading.

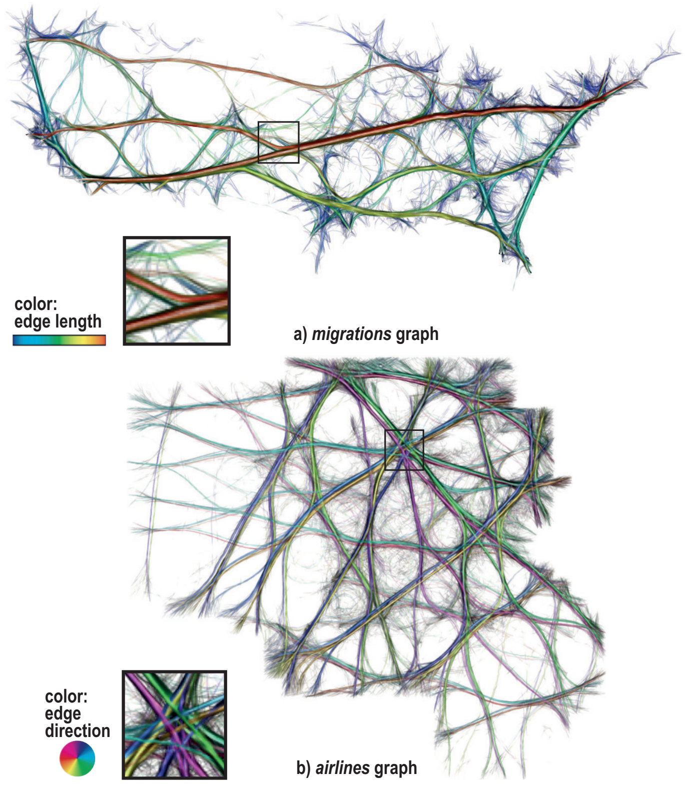


Figure 7.4: Multiscale tube shading for the *migrations* and *airlines* graphs. Insets show shading details (Sec. 7.3.2).

7.4 APPLICATIONS

We next illustrate the added-value of CUBu's scalability and various visual encoding styles by means of several applications involving different types of datasets, questions, and application domains.

Huge graphs: Our first example uses the *amazon* graph, which records about 900K co-purchase relations between about 520K items on *amazon.com* [92]. Figure 7.5 shows this graph, visualized with MINGLE, KDEEB, and CUBu. We see that MINGLE only exposes the edge-density pattern in the graph; KDEEB shows some structure, but cannot outline the main connection patterns. In contrast, CUBu clearly shows these connection patterns. We also see how CUBu generates very similar results for two different sampled versions of this graph, taken from [92] (Figs. 7.5c,e); and how tuning the radius r used for multiscale shading generates coarser *vs* finer-grained visualizations of the graph structure (Fig. 7.5c *vs* d).

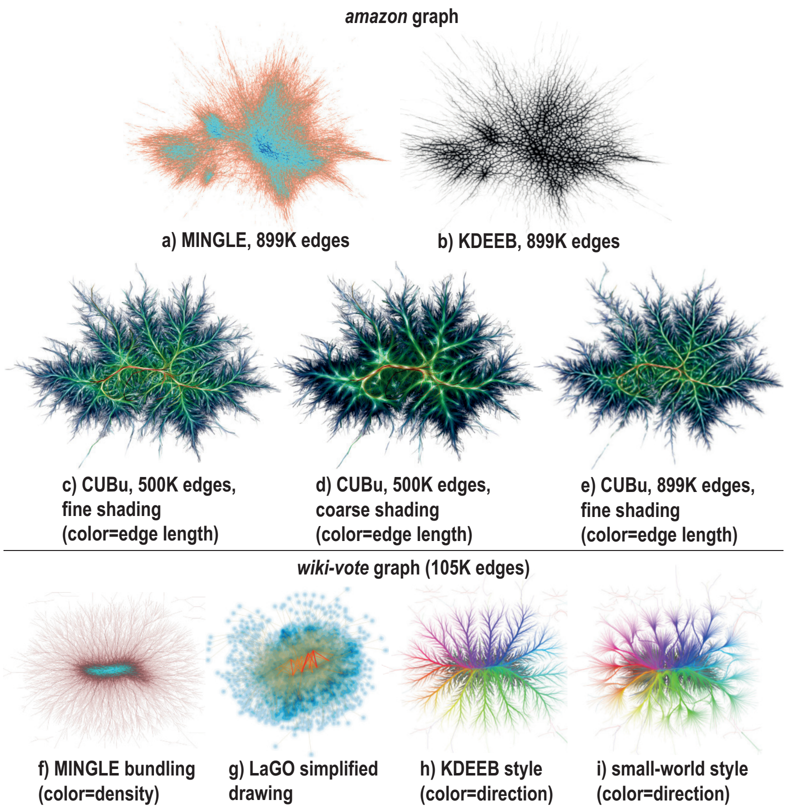


Figure 7.5: Multiscale visualization of large graphs (*amazon* and *wiki*) at different sampling resolutions and shading scales (Sec. 7.4).

Projection analysis: Multidimensional projections are efficient and effective tools for mapping multivariate datasets, having tens or hundreds of attributes per data point or observation, to a 2D scatterplot, so that high-dimensional similarities between points are preserved in this scatterplot [130, 199, 202, 203, 242]. As all existing projection techniques cannot *faithfully* preserve high-dimensional distances, showing erroneously-projected points is crucial to using the resulting projections [168]. One way to spot wrongly-projected points is to draw point-to-point connections (edges) and color these by the projection error [168]. However, this creates a very large, and cluttered, set of lines. Figure 7.6a shows this for a relatively small set of 2300 19-dimensional points from the well-known *segmentation* dataset describing image fragments [88], projected to 2D using the LAMP technique [130]. Here, point-to-point projection errors are color-coded using a rainbow colormap (blue=low projection errors, red=high projection errors). Little structure, in terms of point-groups sharing similar projection errors, can be seen. Using CUBu, we can create a bundled layout of these point-to-point error edges, which now better shows that the main errors affect the top *vs* bottom point groups (Fig. 7.6b). Using our tube-like shading, we now can better spot the top-to-bottom bundle, and also see that an important left-to-right bundle exists (Fig. 7.6c). Finally, using HEB-style bundles allows us to better see how individual projected points participate in bundles, *i.e.*, are affected by projection errors (Fig. 7.6d). All these visualizations are generated in real-time on a commodity PC, due to our fast CUBu bundling technique.

Figure 7.7 shows a more complex usage of CUBu to depict projection errors in a multidimensional projection of a high-dimensional dataset. The projected points, visible as blue dots, are grouped into five clusters, based on their attribute similarity. Each cluster is shown by a colored shaded cushion, using five categorical colors. Bundles show similar points which are placed far away from each other by the projection, *i.e.* the most important projection errors, and are colored by the projection error magnitude. The image is taken from the cover of a recent visualization book [271].

Eye tracking: Our final example shows the eye tracking data set used in [119], which comes from tracking the eye movement over the instruments of an airplane, in a scenario involving a pilot performing a landing manoeuvre in a flight simulator. The aim of this experiment was to test a new cockpit instrument providing landing assistance and its use in combination along the other instruments in the cockpit [119]. In Fig. 7.8 a, this new instrument is indicated as LAI (Landing Assistance Interface). The vertices in the graph are the points to which the eyes where drawn (so-called *fixation points*) and the connecting edges indicate eye-movement between the vertices (so-called *saccades*). For this dataset, fixation points and saccades were obtained from raw eye-tracking data following the protocol detailed in [119].

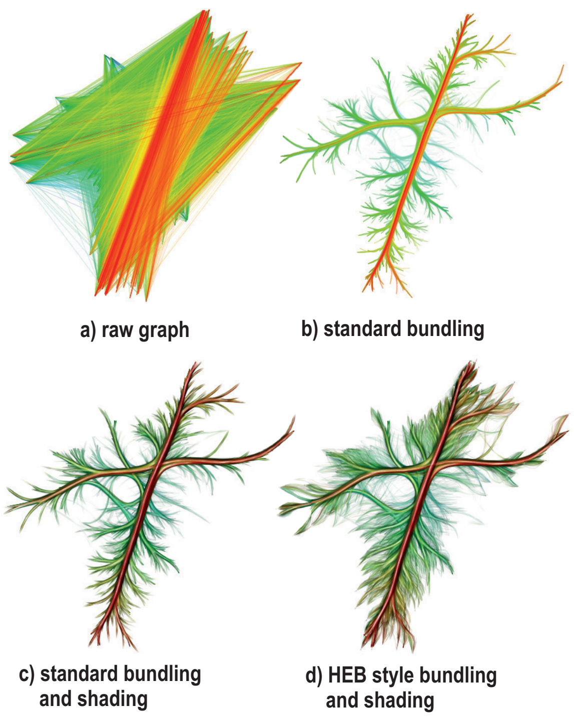

a) raw graph
b) standard bundling
c) standard bundling and shading
d) HEB style bundling and shading

Figure 7.6: Projection errors. Color maps edge lengths (Sec. 7.4).

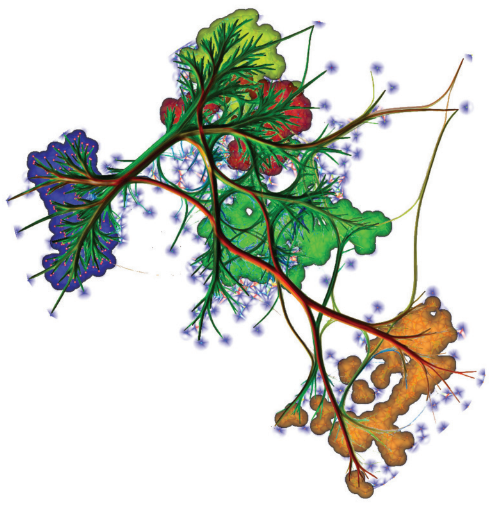


Figure 7.7: Bundled graph showing projection errors between point-groups in a multidimensional projection. Image from cover of [271].

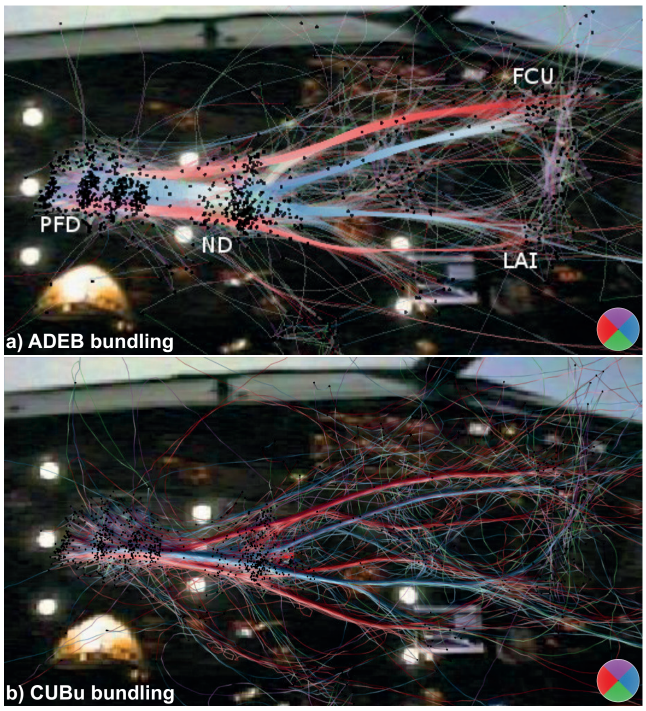


Figure 7.8: Eye tracking analysis of pilot gaze (Sec. 7.4).

Drawing the raw saccades between fixation points generates a completely cluttered image, from which high-level connections between fixation points cannot be inferred. However, bundling can be used to de-clutter and simplify such an image [206]. Figure 7.8 compares this approach using the attribute-driven edge-bundling method (ADEB) [206] and our CUBu method. The figure shows bundles generated by ADEB *vs* those generated by CUBu, both using the same color scheme to show the main direction of the original (unbundled) edges. Like the image generated with the ADEB method, the CUBu bundled image (Fig. 7.8 b) shows the same connections between instruments. However, we can also see a smaller but still significant bundle along the central axis of the image due to the different rendering style allowed by CUBu. Also, CUBu is roughly 100 times faster than ADEB for this dataset (see also Tab. 1).

Flight exploration: A final application of CUBu is the simplified visualization of dynamic flights data, used on the context of helping air traffic controllers with analyzing and planning routes of aircraft [24, 77, 113, 117, 274]. This more involved application deserves more attention, and as such is detailed separately in Chapter 8.

7.5 DISCUSSION

We have implemented CUBu using C++ and NVidia CUDA 2.1 for the bundling part and OpenGL 1.1 for rendering. All bundling steps are fully done on the GPU. Once a bundled result is obtained, the bundled dataset is transferred back to the CPU and rendered by means of standard OpenGL calls.

We have tested CUBu on Linux and Mac OS X with several GPUs (GT 330M, GeForce 580, and single and dual-GPU GTX 690). For the dual GPU, we split all work (density estimation, advection, resampling, and smoothing) evenly on the two GPUs. For testing, we used a variety of graphs, including all static graphs in [92, 108, 116, 168] and the dynamic graphs in [119, 137]. These range from a few hundred nodes and edges to hundreds of thousands of nodes, almost a million edges, and almost 20M edge sampling points (sites).

7.5.1 *Parameter settings*

As visible from the earlier discussion, CUBu offers a quite large flexibility in terms of styles of bundling and rendering. However, this comes at the expense of offering a wide range of parameters to the end user, who should be able to effectively understand them. We discuss these parameters (purpose, effect, range, good preset values) below.

Kernel size p_K: The initial kernel size, specified in pixels, controls bundling coarseness. In detail, p_K tells the user the maximum distance at which two edges 'see' each other, *i.e.* get bundled. Small values yield more, and sparser, bundles; large values yield a simpler view having less and denser bundles. A good preset for p_K is 5% of the size of the graph drawing.

Bundling iterations p_N: The number of bundling iterations should be large enough so that bundling converges to a stable structure. For all tested graphs, we verified that $p_N \in [10, 15]$ leads to convergence, although graphs having already closely spaced edges, such as trail sets [119] converge with less iterations. Hence, we conservatively preset $p_N = 15$.

Image resolution p_I: The output image size controls the accuracy of density estimation, and thus also of gradient estimation and subsequent bundling result. Typical applications would use $p_I = 512^2$ pixels. For high-quality results, such as the images in this chapter, we used $p_I = 1024^2$ pixels.

Sampling points p_S: The edge sampling density, equal to the total number of sampling points divided by the sum of all edge lengths, also affects bundling quality. Intuitively, we want the sampling density to be high enough to capture the smallest details of interest in our graph drawing, but not higher, as this decreases speed. For all our test graphs, we found this

FDEB	DEB	SBEB	GBEB	ADEB	3DHEB	KDEEB	MINGLE	CUBu
65	27.5	26.6	5.6	1.3	0.7	0.5	0.2	0.014

Table 1: Bundling times (seconds) for several methods for the *US airlines* graph (235 nodes, 2099 edges, 86K sample points), see Sec. 7.5.2.

optimal density to be roughly equal to one sampling point per 10 pixels of edge length.

7.5.2 *Performance*

CUBu's performance depends on its four parameters: kernel size p_K, number of bundling iterations p_N, image resolution p_I, and sampling point count p_S. To analyze scalability, we varied all four parameters, one at a time, while keeping the other three fixed around good default values, and measured the bundling time. Figure 7.9 shows our timings on the single and dual-GPU GTX 690 platform, thereby also showing multi-GPU scalability. We see that bundling speed is linear in p_N, p_S, $\sqrt{p_I}$, and roughly independent on p_R. Also, we see that CUBu scales well on a dual-GPU platform. Since our dual-GPU design simply splits workload between the two GPUs, it should also scale well on a platform having more than two GPUs. This is an important result, as none of the bundling algorithms known so far do takes advantage of multi-GPU capabilities.

The complexity of CUBu is $O(p_I p_N p_S)$, worst-case identical to KDEEB and ADEB. Yet, the highly-parallel design of CUBu ensures that it is 30 to 100 times faster than KDEEB, the fastest known undirected bundling competitor (Tab. 2). Compared to ADEB, CUBu is 60 to 200 times faster, as ADEB is half the speed of KDEEB [206]. Further comparison, done on the well-known *US airlines* graph (235 nodes, 2099 edges, 86K sample points), are listed in Tab. 1. These results are not surprising given that the complexities of MINGLE, FDEB, GBEB, and DEB are essentially quadratic with respect to p_S, while the complexity of CUBu is linear with respect to p_S.

The above performance results are difficult to beat. After this work was completed, Lhuillier *et al.* presented FFTEB, a general trail-bundling algorithm that is about two times faster than CUBu for undirected graphs, and about 3 times faster than CUBu for directed graphs, respectively [153]. FFTEB is basically identical to CUBu, except that the computation of the density map ρ (Eqn. 7.1) is performed by multiplication in the Fourier domain, rather than by spatial convolution, in our case. However, this requires additional costs to transform the graph drawing G to the Fourier domain and back, at each iteration step. As noted above, a certain performance increase an be obtained, but not a dramatic one.

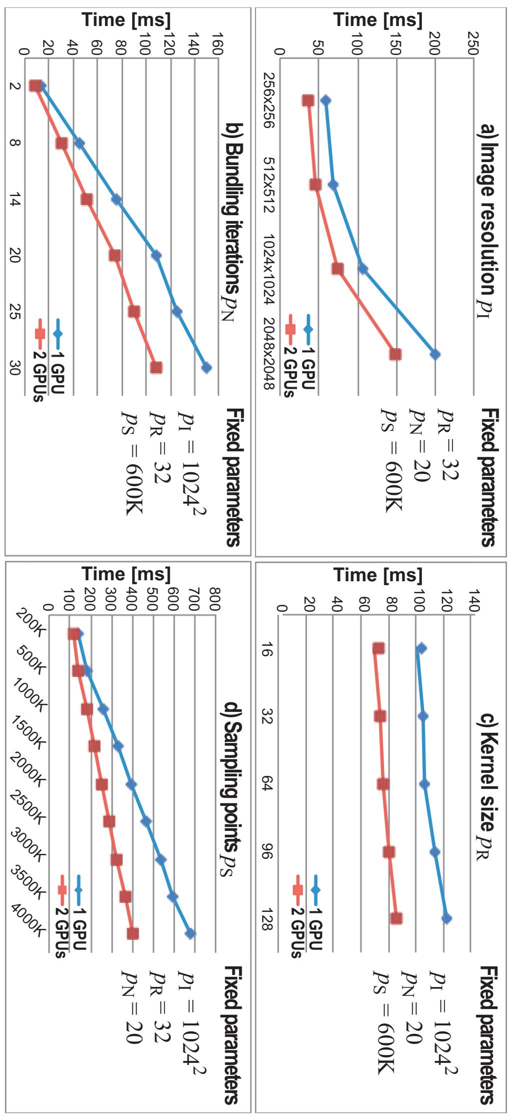


Figure 7.9: Performance scalability as function of the CUBu algorithm parameters (Sec. 7.5.1).

Graph	Nodes	Edges	KDEEB		CUBu (1 GPU)	
			Samples	Time (ms)	Samples	Time (ms)
US airlines	235	2099	86K	500	86K	14
US migrations	1715	9780	220K	1500	221K	24
Radial	1024	4021	290K	1500	290K	23
France airlines	34550	17275	330K	1800	330K	25
Poker	859	2127	50K	400	50K	11
Amazon	738491	899791	19M	8053	19M	152

Table 2: Timings for CUBu and KDEEB [116] for several graphs (Sec. 7.5.2).

7.5.3 *Generality*

CUBu can handle graphs of any topology and having an initial edge layout given by curves or straight lines, as long as we have a 2D node layout. By varying a few parameters, we can achieve undirected or directed bundling and several bundling styles (FDEB, HEB, small-world, and schematic) and shading effects (flat, emphasizing outlier edges, and tube-like) with a single implementation. All bundling parameters, except the image resolution, can be controlled *locally*, by simply making them a function of the data values at any edge sampling point (site) or neighborhood thereof. This way, we can create a rich family of bundling variations. For example, we can set the kernel radius p_K as a function of the parametric site coordinate t to control the bundles' shapes; or we can set the directional compatibility κ as a function of edge attributes, to achieve data-driven bundling. Such variations require only minimal code changes to CUBu and incur no performance penalty, as CUBu treats all sites independently and in parallel.

7.5.4 *Relation to mean shift*

As already outlined, CUBu and its underlying predecessor KDEEB are very similar to computing 2D mean shift (MS) [46] on the edges' sample points. However, there exist some algorithmic differences between these methods, as outlined below.

- MS computes the kernel density estimation (ρ, Eqn. 7.1) only once, at the beginning of the advection process, and next uses a *fixed* gradient $\nabla\rho$ for the entire process. In contrast, we re-estimate ρ and $\nabla\rho$ after each advection step. From our experiments, where we also tried the MS-based approach, we observed that re-estimating the density (as points move) yields a better bundling, which is also less sensitive to the choice of the initial kernel radius R;
- Our advection uses a step whose size does not depend on the gradient magnitude $\|\nabla\rho\|$ (Eqn. 7.2). In contrast, MS uses the actual gradient $\nabla\rho$ as advection step. In our case, we need to normalize the gradient,

since we recompute the density on-the-fly. As such, as sample points get increasingly denser as advection proceeds, $\|\nabla\rho\|$ will also get increasingly larger, which will cause points to move faster as they get closer to each other in the final bundles. This would cause instabilities and major convergence problems. We remove such issues by gradient normalization. Moreover, this allows us an *explicit* control of the advection speed, independent on the local point density, something that MS cannot do.

- During advection, we progressively reduce both the kernel size R and the advection step. MS does not do this – it only uses one kernel size to estimate the initial density, after which the advection step is fully given by the gradient magnitude. We believe our approach to be better: By reducing the kernel size, we effectively obtain a multiscale effect: During the first advection steps, a coarse-scale bundling structure is identified. Next, this structure is progressively refined. In MS, this effect is not possible, since there is a single kernel size. Secondly, our approach guarantees that the advection process converges, regardless of the initial kernel radius R and number of advection steps. Indeed, as explained in Sec. 7.3.1, at the final iteration, the kernel radius R reaches one, meaning that points do not 'see' other points further than a distance of one pixel. Moreover, following Eqn. 7.2, the actual advection step is also one pixel. As such, the advection process is guaranteed to reach equilibrium. Obtaining a similar effect with MS is not trivial, as it requires a careful setting of the initial radius and number of advection steps.

As outlined above, we experimented with both classical MS and our modified version, and obtained (visually) better, and easier controllable, results with the latter for the bundling use-case. This opens the question whether our MS modifications would not be of added value in more general applications such as *e.g.* data clustering or image segmentation. We believe that this is a low-hanging-fruit direction of future research.

7.5.5 *Limitations*

While fast, generic, and highly configurable, CUBu has a few limitations. Like all other bundling methods, its bundles are not fully controllable in terms of *exact* shape and position. Interpreting such bundles should thus be done with care, especially when spatial positions are important. To mitigate this, CUBu adds bundle relaxation [115], which allows users to interactively interpolate between bundled and original edges. Separately, the design of *effective* bundle shapes is clearly application-dependent. The styles shown in Sec. 7.3.1 are just a sample subset which does not claim to be generally optimal nor exhaustive. Specific applications may need different bundle styles. Such styles are easy to get by using other suitable edge profiles (Eqn. 7.5) and/or edge similarity functions (Eqn. 7.6).

A separate technical limitation concerns the scalability of CUBu with respect to the number of sampling points p_S. As noted in Sec. 7.5.2, CUBu scales linearly with p_S. However, an implicit limitation is that the sampled graph must fit in the available GPU memory. Given that, for a sampling point, one typically needs to store two floating-point coordinates, one color attribute, and optionally one floating-point data attribute, a typical GPU having 4 GB VRAM will fit about $p_S = 250$ million sampling points (the costs of storing the density and shading images, as well as those for the windowing system not being accounted for). While this p_S value may seem large, accurately representing a graph like *amazon* (900K edges) at an image resolution of 2000^2 pixels, by having one sample point every few pixels, easily yields over one *billion* sample points. This limitation is solved by FFTEB [153] which extends CUBu by proposing a CCPU-GPU streaming scheme for large graphs, at the expense of speed. They show that this streaming scheme allows bundling graphs of over 60 billion sampling points on consumer-grade GPUs. A possible refinement of this streaming scheme that would increase its speed would be to use an adaptive sampling resolution, where a relatively low number of sampling points p_S is used for the first bundling iterations, and increasingly more sampling points are used for the latter iterations. This would parallel our current coarse-fo-tine multiscale approach where we use large kernels for early iterations and finer kernels for the latter ones.

7.6 EXTENSIONS: ABSTRACT VISUALIZATION OF FLOW FIELDS

Apart from the material published in [315], on which this chapter is based, we have investigated the use of bundling for the simplified (abstract) representation of flow fields. The key goal in this context is similar to the above-mentioned examples where bundling was used to depict a dense set of trails in a simplified, clutter-reduced, manner. The key difference is that, in the current context, the input data is *not* a set of trails, but a vector field which is densely-sampled over a compact spatial region. As such, trails in this case are a *derived* visualization primitive, rather than raw data.

We approach the task of producing simplified (abstracted) representations of vector fields as follows. First, in Sections 7.6.1 - 7.6.6, we introduce the general idea of using continuous visual abstractions to depict vector fields, and present our first solution for this, which does not use bundling. Next, in Section 7.7, we present our second solution to the same task, which uses the CUBu bundling techniques discussed earlier in this chapter, and compare it with the continuous abstract visualization outlined above.

7.6.1 *Context*

To begin with, let us clarify the relation between abstraction and trails bundling.

Abstraction is a—if not even *the*—fundamental principle employed in virtually all areas of visualization because it allows one to uncover and understand principles about the subject matter that we visualize, rather than just seeing the raw data. As Rautek et al.[215] note, abstraction can be introduced in a visualization either implicitly by selecting a certain style of depiction ("low-level visual abstractions") or explicitly by employing means such as focus+context or distortion ("high-level visual abstractions"). The latter group of high-level abstractions are of particular interest because they are created to emphasize specific chosen aspects of interest to the viewers. Bundling, as discussed earlier, is a particular type of distortion-based abstraction which essentially replaces spatially close trails by a single, compact, visual depiction – the bundle. Or, in the light of the mean shift analogy (Sec. 7.3.1), bundling replaces a complex trail-density signal with a simpler one, where high-density regions are emphasized, and low-density ones are essentially removed. Often, however, there exist many different means to achieve explicit or high-level abstractions, all of which are valid and show different important aspects of the same dataset. Therefore it is essential that we can link these different types of abstraction with each other [66] to allow viewers to understand the relationship between them.

As explained in the previous sections, CUBu provides a multiscale view on a trail-set, where the scale parameter is essentially controlled by the KDE kernel radius R (Sec. 7.3.1). However useful, such visualizations have the *same* level of simplification everywhere, as R is a global parameter. Moreover, even for different R values, used *e.g.* in different images, the same *representation* is used for the trail-set, which consists of a set of curved and shaded trails.

Different design decisions can, however, be used to create a multiscale visualization. In the following, we examine the idea of combining multiple different layers of abstraction, or scales, of the same dataset. We specifically consider datasets whose abstract representations are *spatially* and *semantically* nested. By spatially nested, we mean that such abstractions have the same spatial embedding but each uses a different amount of screen space, so that more abstract representations are, generally, 'inside' less abstract ones. This generalizes the bundling principle where a bundled set of trails is, typically, located inside (to be more precise, is centered within) the spatial extent of the corresponding set of unbundled trails.

Separately, our input data to visualize is also more general than trail-sets. Specifically, we consider vector fields defined over compact spatial regions $D \subset \mathbb{R}^3$. One of the representation choices for such vector fields is as a set of streamlines, densely seeded, and densely covering, D. These are essentially nothing but trail-sets, much like the airplane or eye-track trails discussed in Sec. 7.4. Given their 3D embedding, such dense streamline sets exhibit very similar clutter and overlap problems as their 2D counterparts discussed earlier. As such, one of the goals of our simplified visualization (reducing such problems) is identical to the key goal of bundling. However, there is a key difference: For all the examples presented so far, the considered trail-sets were the *actual* data that we wanted to visualize. For 3D vector fields, streamline sets are just one of the possible *representations* one can choose for the actual vector field data.

The representation nesting property mentioned above allows us to use two-dimensional techniques to generate real-time halos which appear volumetric and visually separate the different flow field visualizations. This allows us next to create transitions between different abstractions which do not naturally allow seamless interpolation as demonstrated using different representations of fluid flow. We also incorporate lens-based navigation into the defined abstraction space allowing investigation of additional intermediate abstraction levels. Taken together, these techniques facilitate an intuitive continuous navigation of a set of nested abstractions of a given 3D flow visualization. This is analogous in spirit to changing the bundling kernel-radius parameter R, or the relaxation factor (see Sec. 7.3.1), but very different in terms of design. That is, in both cases, we achieve a visually continuous change of the depicted image from a more complex to a simpler one; however, for bundling, this is done using the same visual metaphor, instantiated with different parameter values. For the techniques presented next, this is done using a mix of *different* visual metaphors.

7.6.2 *Visually abstracting data*

Abstraction is a core principle in visualization and takes many forms, depending on the visualized data. Dedicated controlled abstraction has been investigated not only in non-photorealistic rendering (e. g., [54, 179, 295]) but also in visualization. In the field of information visualization many forms of intentional abstraction are used, the main ones being edge bundling (already extensively discussed) and focus+context (e. g., [35]). In scientific and specifically illustrative visualization many "high-level visual abstractions" [215] are used.

Relevant for our own work are those high-level visual abstractions that not only show more or less relevant parts of a dataset in more or less detail, but which can relate different visual representations (i. e., different abstraction levels) to each other. Duke [66] describes this problem nicely and suggests linking different types or representations to each other to uncover and understand the structure of a dataset, naming molecular visualization as one example. In a separate work, we have demonstrated such seamless transition between molecular abstraction levels in an interactive [316] and a spatially explicit [161] manner. However, our realization of abstraction level transitions in [316] requires that meaningful intermediate stages exist. This is not the case for many forms of abstraction — a problem that we address next.

We use halos as one visual technique to visually separate layered elements and thus enhance spatial perception. While halos enhance the spatial perception of the depicted objects in volume [31, 126] and other visualization domains [79, 262], we use them to support visual layering and thus related to their function of showing occlusion relationships in line-based techniques [9, 70]. In addition, we use interactive lenses to locally explore our layered abstractions. Lenses are not only frequently used to support focus+context techniques [35] but also to interactively reveal otherwise hidden information, an approach pioneered by Bier et al.[22, 23]. Their so-called Magic Lenses locally affect a 2D screen region using a user-selected operator. While lenses can be used in a 3D context to distort the projection [303], they can also be used to specify non-view changes for a 3D scene in a separate 2D layer [123, 186]. Lenses have also been used for exploring bundled trails by essentially restricting the bundle relaxation factor with respect to the lens position [114]. Our lenses have a similar function as we use them in a 2D layer over the 3D model to locally reveal relationships between abstraction layers, thus also relating to a number of so-called 'smart visibility' methods in visualization [285].

7.6.3 *Visualization Model*

We start with a dataset $d \in D$, where $D \subset \mathbb{R}^3$, and consider several visualizations of d, modeled as images $V_{1 \leq i \leq N} : D \rightarrow \mathbb{R}^2$. A visualization V_i can be seen as a function that takes d to produce a 2D image $A_i = V_i(d)$. We call these images *abstractions* of d if they represent the information in d on different levels of detail. We distinguish two abstraction types: *Semantic* abstractions A_i simplify the information in d by showing varying amounts of the information present in d using different visual representations. For example, a fluid flow volume $d \subset \mathbb{R}^3$ can be rendered as an entire flow volume using LIC [32, 250], as stream LIC structures for a set of given streamlines [105], and as flow topology [275]; these are increasingly simplified semantic representations. Bundling is also a semantic abstraction. *Sampling* abstractions reduce the amount of points produced by a given semantic abstraction A_i using data sampling. Rendering different numbers of streamlines, for example, are samplings of the streamline abstraction. We denote all S_i samplings of a semantic abstraction A_i by $A_i^j, 1 \leq j \leq S_i$ with $A_i^1 = A_i$ the most detailed and $A_i^{S_i}$ the coarsest sampling.

To be useful in an exploration scenario, abstractions must be described in terms of the amount of *simplification* they produce on some input dataset. In our model we assume that, for a given application domain (e. g., flow visualization), the abstraction set $A = \{A_i\}$ can be ordered in decreasing amount of provided simplification from the densest abstraction A_1 to the sparsest one A_N. We also require that simpler abstractions are visually *nested* within less simple ones, i. e. $A_j \subset A_i, \forall i < j$. This is often the case in scientific visualization where abstraction reduces the size and/or spatial dimensionality of the depicted visual elements while keeping them aligned in the space D. Our flow visualization scenario is such a case of nested abstractions: the topology is a part of the streamlines, these are nested within stream LIC representation, which in turn is a part of a LIC volume. Bundling also obeys the nesting property – a bundled image is spatially located within the extent (hull) of the unbundled image. Another example of nesting is the multiscale representation of 3D shapes by means of medial axes or skeletons [217]: A shape can be visualized by drawing its actual boundary (A_1), its 3D medial surface consisting of a set of manifolds (A_2), its curve skeleton (A_3), and its barycenter (A_4) are increasingly simple, and nested, semantic abstractions. Reducing the number of points on each such abstraction by means *e.g.* of surface decimation or skeleton pruning creates additional nested sampling abstractions.

7.6.4 *Navigating the Abstraction Space*

Given an abstraction set A as just described, one typically wants to *navigate* A to get different types of insight which are best visible at different abstraction levels. One navigation option is to start with A_N (most abstract) and browse through A_i until A_1 (most detailed), optionally using spatial sampling to restrict the dense-data areas to zones of interest using, e. g., focus+context techniques. One can also start with the most detailed level A_1 and simplify the visualization to the coarsest level A_N is reached. During both navigation types, we call the level of the highest abstraction A_f being visualized the *focus* of the visualization: Given a user-selected f, we aim to produce a visualization combining all *context* abstractions $A_i, i < f$ and A_f in a single visualization such that all abstractions and their spatial nestings are shown. This will permit smooth navigation in the *combined* space of semantic and sampling abstractions (as introduced in subsection 7.6.3).

Such navigations are typically realized by toggling the rendering of the elements A_i on and off. However, this creates sharp visual discontinuities in the transition, especially if the abstractions differ visually. Continuity can be added by smoothly interpolating the transparency or shape of consecutive A_i using fading or morphing while navigating through A. Bundling can be seen in this light too: The unbundled data A_1 can be continuously morphed to the fully bundled version A_N by changing the so-called bundle relaxation parameter with N steps. However, interpolation here between the abstractions A_i is done along the *temporal* dimension, while the context we are examining here regards interpolation in the *spatial* dimension. Blending blurs the spatial nesting insight and can result in too high opacities when too many abstractions are blended. Morphing is not trivial for any pair of (nested) shapes, works only for shape pairs, and requires 3D shape representations rather than their 2D visualization results A_i.

We propose to create a continuous navigation function $Nav : A \times [0,1] \to \mathbb{R}^2$ to help navigation in the abstraction space. Given our set A of ordered, nested abstractions and a focus abstraction level $f \in [0,1]$, we combine all abstractions A to build a visualization V. As the user changes the focus level f, V continuously changes to show only A_1 (at $f = 0$), next show the focus abstraction A_f nested within lower abstractions as context, and finally A_N (at $f = 1$). The design of Nav should be such that it can be computed on any set of nested 2D or 3D abstractions, is continuous in f, clearly emphasizes the focus-context relation of nested abstractions, and is computed using only 2D image information instead of 3D shape information to achieve maximal performance.

We use an additive blending of the abstractions A_i in nesting levels (decreasing i) and compute the navigation function as $Nav(A, f) = \sum_{i=1}^{n} \alpha_i(f)A_i$ (see Figure 7.10). The design of the blending factors $\alpha_i : [0, 1] \to [0, 1]$ is essential, we use $\alpha_i(f) = \phi_i(f) \cdot \psi_i(f) \cdot h_i(f)$. Here, $\phi_i : [0, 1] \to [0, 1]$ is the function used to fade in an abstraction A_i, $\psi_j : [0, 1] \to [0, 1]$ is the function responsible for fading out a (context) abstraction $A_j, j < i$, and $h_i : [0, 1] \to [0, 1]$ is the *halo function* used to create a halo around the selected abstractions. As fade-in function we use $\phi_i = \max\left(0, \min\left(1, \frac{f - f_{in}^{(i)}}{f_{full}^{(i)} - f_{in}^{(i)}}\right)\right)$ where $f_{in}^{(i)}$ is the focus value from which we start fading in abstraction A_i and $f_{full}^{(i)}$ is the value at which A_i is completely visible. In practice, we want to start fading in the next abstraction A_{i+1} when the current abstraction A_i is fully visible. Hence, we choose $f_{in}^{(i+1)} = f_{full}^{(i)}$. Similarly, we define the fade-out function $\psi_i = \min\left(1, \max\left(0, \frac{f_{gone}^{(i)} - f}{f_{gone}^{(i)} - f_{out}^{(i)}}\right)\right)$. Here, $f_{out}^{(i)}$ is the value for which we start fading out the context abstraction A_i and $f_{gone}^{(i)}$ is the value for which abstraction A_i is no longer visible. Similar to fading in, we start fading out abstraction A_{i+1} when the A_i is no longer visible and, thus, set $f_{gone}^{(i)} = f_{out}^{(i+1)}$. Also, we constrain $f_{out}^{i} > f_{full}^{i+1}$ such that A_i does not start fading out before A_{i+1} is fully in focus. Finally, we set $f_{full}^{1} = 0$, $f_{out}^{N} = 1$ so that we start with a fully focused A_1 and end with a fully focused A_N.

While combining ϕ_i and ψ_i allows us to fade abstractions in and out of view continuously, the resulting image will be unable to clearly show the *nesting* structure of the abstraction space: Depending on the specific abstraction image shapes and colors, it may be hard to see which result pixels belong to a (thin) abstraction being nested within a (larger) context abstraction, especially if both have similar colors. We therefore use the *halo function* h_i to generate halos around abstractions: $h_i(f) = \min\left((DT_{A_{i+1}}/\delta)^{k_i(f)}, 1\right)$. In this function, $DT_\Omega : \mathbb{R}^2 \to \mathbb{R}^+$ is the distance transform of a 2D binary shape Ω, which gives, for any points $\mathbf{x} \in \mathbb{R}^2$, its distance to Ω [51]. DT is zero inside Ω and smoothly increases outside the shape. In our case, we construct such shapes by simply thresholding the rendered abstractions A_i into foreground (rendered) and background (not rendered) pixels. Having $DT_{A_{i+1}}$, we compute a halo around A_{i+1} by modulating the distance transform with a power function $k_i(f)$. The halo's width is limited to a maximal value of $\delta > 0$ pixels. The effect of the power function is to create a smoother transition from the context A_i of A_{i+1} than if linear distance functions were used. Finally, we set $k_i = \phi_i$ to increase the halo around the fading-in abstraction A_{i+1}, thus making it more prominent in its context A_i where it is nested. Perspective-like halos can easily be obtained by modulating the value of δ with the depth of A_{i+1} at each pixel.

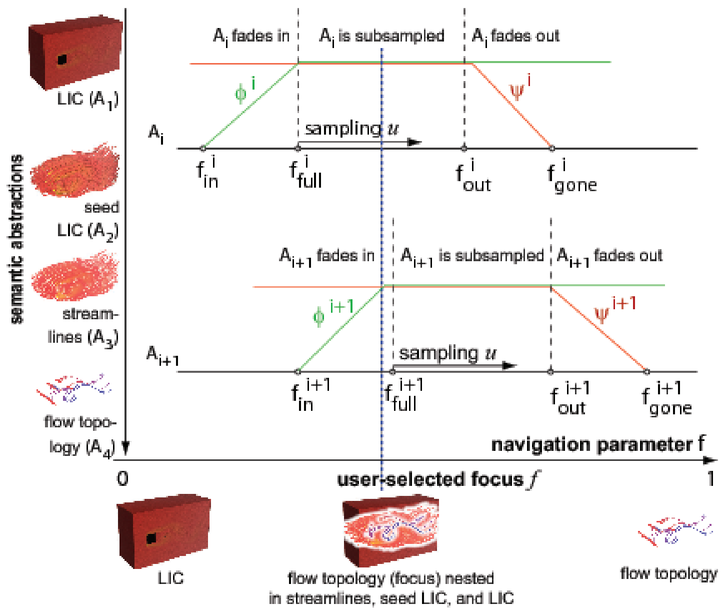


Figure 7.10: Continuous navigation in a flow visualization abstraction space with four abstractions A_1–A_4.

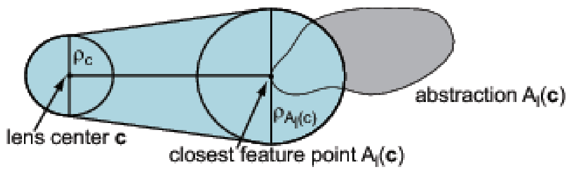


Figure 7.11: Construction of the focus guided lens. Typical values are $\rho_c = 5$ pixels and $\rho_{A_l(c)} = 90$ pixels.

The above process describes how *semantic* abstractions A_i are combined into a single image. However, as defined in subsection 7.6.3, our input may contain sampled versions thereof. We integrate these smoothly in the above process by replacing, in the navigation function $Nav(A, f)$, each semantic abstraction A_i with its sampled version A_i^j, the sampling parameter j being controlled by the distance from the user-set focus f to the full-visibility f^i_{full} as $j = \frac{f-f^i_{full}}{f^i_{out}-f^i_{full}} S_i$. In other words, as the user increases f, the full-visibility abstraction is progressively simplified from A_i^1 to $A_i^{S_i}$ (coarsest variant). During this process, all remaining visualization elements stay the same (halo sizes, overall abstraction transparency). When f reaches f^i_{out}, the coarse abstraction $A_i^{S_i}$ is further faded out. For abstractions which have no level-of-detail representations, the process simply uses the unique representation A_i. This directly accommodates any number of semantic abstractions with any number of sampling representations thereof, effectively intertwining the navigation in the semantic and sampled spaces of abstractions.

7.6.5 *Interactive Local Exploration*

While this navigation facilitates an effective *global* abstraction space exploration, in many cases we are interested in getting local detail information. We thus also provide context-sensitive *local* lenses, whose goal it is to allow parts of an abstraction $A_{l>f}$ located inside the lens to become visible even when otherwise hidden due to the global abstraction level f. While a naïve implementation would change the blending factors $\alpha_i(f)$ close to the lens center, this would interfere with our transparent distance-based halos. In multi-layer visualizations such as ours one also wants to see 'deeper' within the abstraction stack inside the lens *and* locate the parts of deeper-nested abstractions closest to to the lens.

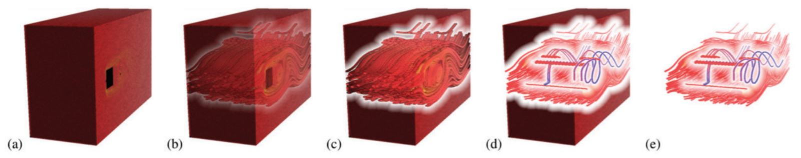


Figure 7.12: Navigating the flow visualization abstraction space: (a-c) Introducing seed LIC as focus in the LIC volume, (d) has the flow topology as focus abstraction and the previous abstractions as context from which the LIC volume is removed in (e).

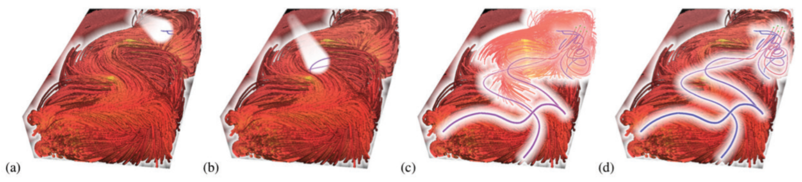


Figure 7.13: Fluid flow abstracted (a-b) locally with a guided lens and (c-d) with sampling abstractions (streamline filtering).

For this, we use a *focus-guided lens*. Given a global abstraction level f, we combine revealing deeper-nested information at the lens center with revealing higher-abstraction structures $A_{l>f}$ close to it. We first locate the closest point $A_l(\mathbf{c})$ of abstraction A_l to the lens center $\mathbf{c}$. The point $\mathbf{c}$ can be directly computed as $A_l(\mathbf{c}) = FT_{A_l}(\mathbf{c})$. Here, $FT_{A_l} : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ is the *feature transform* of the shape A_l [51]. The feature transform of a shape $\Omega \subset \mathbb{R}^2$ is defined as $FT_\Omega(\mathbf{x}) = \{\mathbf{y} \in \Omega | DT_\Omega(\mathbf{x}) = \|\mathbf{x} - \mathbf{y}\|\}$, i. e.the closest point $\mathbf{y} \in \Omega$ to a given target point $\mathbf{x}$, restricting ourselves to a one-point feature transform [75]. With the lens center $\mathbf{c}$ and closest abstraction point $A_l(\mathbf{c})$, we construct our focus-guided lens by multiplying the halo functions $h_{l>f}$

with the distance transform of a beam-like shape created by two circles connected by a trapezium (light blue in Figure 7.11). As the lens is moved, it behaves similarly to a light beam that shows the shortest spatial path from the lens center to the desired A_l. This is useful as one does not need to fully remove (make transparent) *all* abstractions $A_{k<l}$ in order to discover A_l. Hence, one can stay at a desired semantic focus level f and use the lens to search for another desired $A_{l>f}$ in the vicinity of any point.

7.6.6 *Implementation and Results*

For the realization we only require a set of $N-1$ 2D images depicting the different context abstraction levels and one image depicting the abstraction in focus at the selected abstraction level, as our method works entirely in image space. These images are either generated on-the-fly or are precomputed if they do not change during the exploration. From these images we compute the soft halos for our nesting (within 10 ms on a modern graphics card) by employing a recent CUDA-based implementation [265] of exact Euclidean distance transforms and feature transforms [34]. Finally, blending is implemented via OpenGL alpha blending. The entire process, including rendering LIC, seed LIC, streamlines, and precomputed topology, works at 5 frames per second on a MacBook with 2 GB RAM and an NVIDIA GeForce 320M graphics card with 256 MB RAM.

We present results created using our technique by providing an example use case scenario. We use a data set which results from a direct incompressible Navier-Stokes simulation of the flow around a cylinder with a resolution of $100 \times 60 \times 20$ grid cells. A researcher interested in studying this data set might start with looking at the LIC visualization (the least abstracted representation, shown in Fig. 7.12a). Increasing the abstraction level f results in the next abstraction (the streamline LIC structures) being slowly blended in, generating a halo around this new focus abstraction to make it visually distinguishable from the LIC visualization which is now the context abstraction. Images (a)-(c) provide the result of several steps in this transition, resulting in an image in which the streamline LIC abstraction can be studied while the least abstracted visualization (LIC) is used to provide context.

Our second example concerns the visualization of the results of a different incompressible Navier-Stokes simulation, with a resolution of $128 \times 85 \times 42$ grid cells, where flow enters the domain through a small inlet and flows around two planes. Fig. 7.13a shows the global behaviour of the flow using the SeedLIC abstraction as focus and the LIC volume as context and the lens used to explore the highest level of abstraction (see Sec. 7.6.2). This lens always indicates the closest point of the selected abstraction as also shown in Fig. 7.13b. Figure 7.13c shows the streamlines in focus with the LIC volume as context. Increasing the focus f not only blends in the next abstraction (the seeded streamlines) but also reduces the amount of streamlines away from the flow inlet. While less streamlines are shown, the soft halos still provide a good separation between the streamlines and the seedLIC abstraction in which the streamlines are nested. Increasing the abstraction level f further removes all streamlines from view, resulting in an image where we can study the realation between the seeded streamlines and the seedLIC structure while still using the LIC volume as extra context (Fig. 7.13d).

7.7 SIMPLIFIED FLOW FIELD VISUALIZATION VIA BUNDLING

As outlined in Sec. 7.6.1, bundling can be seen as a form of abstraction. As such, it is interesting to compare its results with the continuously-nested visualization abstraction proposed for flow fields in the above sections, on the same dataset.

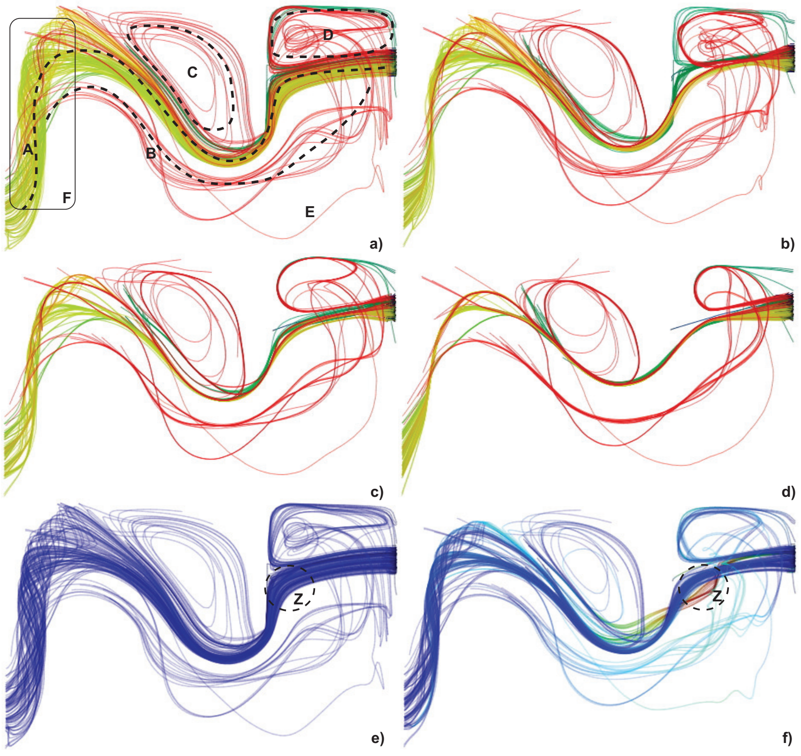


Figure 7.14: Fluid flow abstraction using streamline bundling. (a) Unbundled streamlines. (b-d) Progressively bundled streamlines. (e,f) Visualization of bundling distortions. See Sec. 7.7.

However, to do this directly, we would need to avail of a bundling method able to handle 3D trails, given that our flow datasets shown in Sec. 7.6.6 are volumetric. While implementing such an extension of CUBu is possible, in theory, this is quite hard to achieve in practice. Key to this problem is the efficient computation of the density map ρ (Eqn. 7.1). As explained in

Sec. 7.3.1, this is done using a gathering strategy implemented using CUDA 2D floating-point textures. To extend this to 3D, we would need large-scale 3D floating-point textures. While such textures are supported in CUDA, the speed of the corresponding KDE estimation would be significantly lower than the 2D bundling algorithm we have presented. Additionally, memory constraints would prevent us from using volumes larger than roughly 512^3 voxels on current consumer-grade GPUs.

Given the above, we approached the problem of bundling 3D streamlines by considering a two-dimensional view thereof. For this, we ignore the third coordinate of the streamline sampling points, which yields a set of 2D trails, that we can directly bundle with CUBu.

Figure 7.14 shows the obtained results. The first image (a) shows the raw, unbundled, streamlines, colored by streamline length, using the same colormap as in earlier figures, *e.g.*, Fig. 7.2g,h. The dataset used is the same as in Fig. 7.13, as seen from above. Imagess (b-d) show three progressively simplified flow views, obtained by using increasingly large kernel radii K for bundling (see Eqn. 7.1). As visible, the flow structure is abstracted into its key elements – the two locally laminar flow areas A and B, and the two vortices C and D, indicated by dashed annotations in Fig. 7.14a. Moreover, remote outliers, such as streamline E, are still visible in the bundled image, since the radius K limits the spatial extent of the simplification. However, local flow details, such as the helix-like flow region F, are quickly disappearing in the simplified visualizations.

If we compare these results with those produced by our continuous-nested method, called next CNM for brevity, see Fig. 7.13, several points appear. First, the bundling simplification has a *global* nature, and cannot allow several levels of abstraction to co-exist in the same image; this is possible by construction in the CNM method. However, bundling offers a continuous nesting in the *temporal* dimension, by allowing one to interactively change the kernel radius K and view different abstraction levels. In turn, bundling does not need to cope with the problem of combining different visual abstraction levels, since there is only one visible at a time. This challenge does exist for CNM, and it is solved by halos and blending. Finally, CNM offers an interactive local level-of-detail tool. Our images in Fig. 7.14 do not show this. However, such tools are easily implementable for bundled visualizations, as discussed in detail in [114]. Summarizing the above, trail bundling can be seen as a *supplementary* level of abstraction in the nested hierarchy discussed in Sec. 7.6.4, as it has all the required properties (simplification, continuity, and spatial nesting). As such, it could be incorporated in the CNM visualization solution we have presented so far in this section.

However, as compared to CNM, bundling has the property that it *distorts* streamlines as it simplifies the visualization. This can lead to undesired effects in terms of the type and correctness of insights that one obtains from the bundled visualization. For instance, looking at Fig. 7.14b-d, one cannot (easily) assess that there is a helix flow in region F, while such a flow is evident in the unbundled data or in the CNM visualizations (Fig. 7.13). The issue of bias introduced by bundling pertains to all applications where one bundles trails (rather than abstract connections in a graph), including the flight visualizations and eye trails discussed in Sec. 7.4.

We propose here two techniques to alleviate this issue. First, we allow the user to specify a maximal displacement d_{max} for all points of a bundled drawing. For this, we essentially modify Eqn. 7.2 to include a test that compares the current position of an advected sample point $\mathbf{x}_j$ with its original position $\mathbf{x}_j^{orig}$ in the unbundled data. The advection is then stopped (ignored) if $\|\mathbf{x}_j - \mathbf{x}_j^{orig}\| > d_{max}$. Secondly, we color the bundled trails by the displacement of each point $\mathbf{x}_j$ with respect to its counterpart $\mathbf{x}_j^{orig}$ in the input dataset. Figures 7.14e,f illustrate these techniques. The image in Fig. 7.14e shows the original unbundled streamlines of our 3D flow dataset, colored by displacement, using a rainbow colormap. Obviously, since we show the unbundled data, there is no displacement. Next, we set d_{max} to a small value and bundle the data to obtain Fig. 7.14f. Compared to the (highly) bundled images (b-d), we now see, indeed, there is far less deformation. Moreover, we can find the areas where we can highly trust the data by looking for the blue zones, and areas where the deformation was maximal by looking for the red zones, respectively. As visible in the image, there is a single area where a high deformation level occurred (Fig. 7.14f, marker Z). Visually comparing this area with the unbundled data (Fig. 7.14e) shows, indeed, that there is a high deformation here. This technique is simple to implement, works real-time, can be easily incorporated in any bundling method, and allows one to easily see *and* control where, and how much, bundling deformations occur.

7.8 CONCLUSIONS

In this chapter, we have addressed the visual analysis of large sets of spatial 2D and 3D trails. To this end, we focused mainly on the reduction of visual clutter caused by overlap when visualizing such large trail sets. To reduce visual clutter, we investigated two approaches. First, we introduce CUBu, a general-purpose framework for creating high-quality bundlings from very large graphs. CUBu proposes a GPU-based design that addresses the main desirable features of existing bundling algorithms (scalability, directional bundling, level-of-detail visualization of bundled results) in a single unified algorithm. CUBu is 50 to 100 times faster than state-of-the-art bundling methods, thereby opening the door to real-time bundling of graphs of millions of edges. Separately, CUBu can produce bundling styles similar to a wide variety of existing graph visualization algorithms, such as hierarchical edge bundling, skeleton-based edge bundling, force-directed edge bundling, schematic graph drawing, image-based edge bundles, and dynamic-graph bundling. We compare CUBu with seven related bundling algorithms and show its scalability and generality on several graphs and trail-sets up to one million edges. Separately, we propose a continuous visual abstraction method for 3D trail sets which consists of several nested visual representations. We explain how such representations can be constructed, and how visual continuity can be achieved between them, both locally and globally, by using a set of interpolation, shading, and interaction techniques. We demonstrate this technique for the simplified visualization of 3D flow fields, and also compare it with the simplification achieved by bundling for the same type of data, rendered as streamlines.

Several directions of future work exist. Related to bundling, it is definitely interesting (and useful) to investigate the bundling of 3D trail sets. As outlined in Sec. 7.7, we cannot still do this, and we use for such datasets a dimensionality-reduced proxy, where one of the spatial dimensions is dropped. For our concrete use-case and dataset discussed there, this reduction had limited effect, as the dataset does not exhibit a high extent or large value variations along the dropped dimension. However, in general, 'deep' volumetric datasets exist, and for these bundling should be performed natively in 3D. Exploring how to extend CUBu to efficiently handle such datasets is a first challenge. With this in place, one can then next explore novel ways to produce simplifications of diffusion tensor images (DTI) using bundling, by extending and improving existing work in this direction [27].

Apart from the above, and as a concrete larger-scale application of CUBu than the examples already covered by this chapter, we will show next in Chapter8 how the fast and scalable bundling offered by CUBu enables the construction of interactive exploratory visualizations for very large time-dependent trail datasets, for the study of spatio-temporal patterns of aircraft trails. This work also fits within another general extension direction, namely studying how bundled visualizations can display multiple attributes per trail and/or trail point, so as to handle multivariate datasets.

8 BUNDLED DYNAMIC VISUALIZATION OF FLIGHT DATA

In Chapter 7, we have introduced CUBu, a fast and scalable method for generating simplified visualizations of large graph or trail drawings. As shown by the various examples introduced at that point, CUBu can effectively and efficiently generate images of such datasets where clutter is (significantly) traded off for overdraw. This simplifies the resulting images, allowing one to detect easier large-scale connectivity patterns in the underlying data, such as groups of edges in a graph that link densely packed sets of nodes, or groups of closely running trails in a trail-set. Additionally, we have shown how extra information can be encoded atop of the resulting simplified (bundled) images by suitable color mapping and shading. However, the bundling work presented in Chapter 7 have two limitations:

1. the bundling techniques in Chapter 7 handle only *static* trails, with no or maximally one attribute (direction). Showing whether and how CUBu can handle dynamic and multivariate trails is still an open question;

2. the bundling examples in Chapter 7 focuses mainly on illustrating the *technique*'s capabilities. Seeing how CUBu actually helps in an end-to-end *application* is still to be shown.

This chapter focuses precisely on covering the above two issues[1]. To this end, we choose an application where large trail-sets are the key data – the visual exploration of massive datasets containing flight trails. The users in focus for this type of application are Air Traffic Control (ATC) experts, who are interested in quickly getting overviews of large trail-sets and also using visualization to pose and answer specific queries involving the flight data. To make the problem more challenging with respect to the data size, we consider also dynamic datasets, *i.e.* trail-sets in which the contained trails have timestamps indicating both their lifetime as well as the instantaneous positions of the respective aircraft. We show how the trail bundling offered by CUBu can be used to create interactive explorations of such large spatio-temporal trail sets. Additionally, we show how the basic visual encodings proposed in Chapter 7 can be extended to show extra information which is relevant to the ATC tasks at hand.

1 This chapter is based on the publication: T. Klein, M. van der Zwan, and A. Telea. Dynamic multiscale visualization of flight data. In S. Battiato and J. Braz, editors, *Proc. of the* 9th *IEEE International Conference on Computer Vision Theory and Applications (VISAPP)*, volume 1, pages 232–240, 2014.

8.1 PROBLEM CONTEXT

In the last years, the availability of large and accurate data sources describing the motion of various types of vehicles, *e.g.* airplanes, vessels, automobiles, and pedestrians, has massively increased [6–8, 136, 207]. The availability of such movement datasets can help in a wide range of analyses and use-cases, such as Air Traffic Control (ATC), epidemics propagation, and crisis situation analysis.

All above datasets can be described essentially as a set of spatio-temporal *trails*, *i.e.* paths of moving objects, annotated with both location and time information. Within this context, we focus next on the analysis of airplane movement datasets (other types of vehicle trails can be treated similarly). Such datasets consist of several airplane trajectories, or trails, each one being in turn a temporal sequence of data points describing the position, height, velocity, flight direction vector (and possibly more attributes) of a single airplane over its flight time span. Visualization of flight trails can assist in numerous ATC scenarios, such as finding and explaining historical flight outliers; understanding the correlation between flight congestion and weather patterns; training of ATC controllers; and better planning of flight routes over given spatio-temporal intervals [24, 77, 113, 117, 274].

However, visualizing large trail datasets poses several challenges, of which we consider here the following:

Computational scalability: Movement datasets are by their nature orders of magnitude larger than their static counterparts. For instance, Fig. 8.1 shows a single day of air traffic over France, which contains 20K trajectories, each having hundreds of data points (one data point is recorded every 4 minutes). The trail datasets in Chapter 7 are of a similar order of magnitude – for example, the dataset used in Fig. 7.1 has also approximately 20K trajectories. However, a dataset capturing the air traffic over the entire world and over several weeks will easily have millions of trails. Generating real-time visualizations from such datasets is clearly a computational challenge.

Visual scalability: Besides the computational challenge, large trail datasets will also contain many high-density traffic regions. In turn, visualizing such regions will create visual clutter and occlusions. Moreover, if we want to depict not just spatial positions, but additional attributes such as speed, flight ID, and flight height, the information density increases even further. Finally, if we want to focus on the study of the *dynamic* properties of the data, such as showing how airplane trajectories change over days or weeks, simply using a static visualization that shows all trails is clearly not optimal.

In this chapter, we present a visualization system for air traffic that aims to address the above challenges. In contrast to ATC systems that address more specific use-cases [77, 94, 113, 274], our goal is to efficiently and effectively visualize attributed trails over large time and space inter-

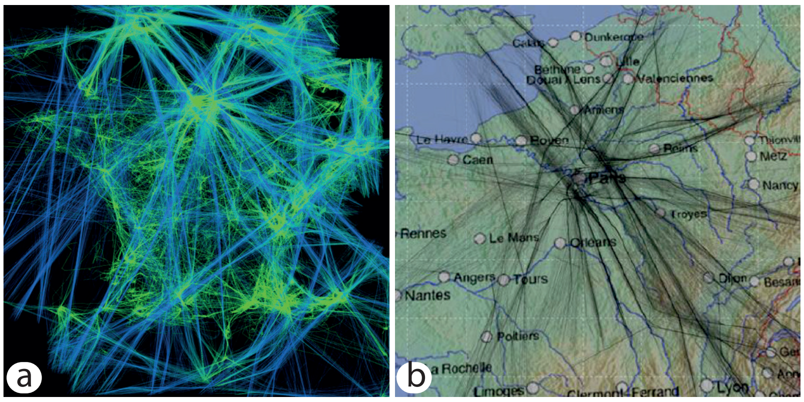


Figure 8.1: (a) Flights over France, July 5th, 2006, visualized with [113], color-coded by height. (b) Zoom-in over Paris area. Compare to Fig. 7.1.

vals. We achieve visual scalability by several level-of-detail, or multiscale, techniques: bundling, animation, and density maps. We achieve computational scalability by implementing all above techniques efficiently on the GPU. Overall, our contribution extends earlier work in trail visualization [113, 117, 225], and particularly our CUBu technique from Chapter 7, with several temporal attributes, on the one hand, and making the visualization suitable for large dynamic datasets, on the other hand. We demonstrate our visualization on both medium-scale datasets (French air traffic, one week) and very large datasets (the world, one month).

The structure of chapter is as follows. Section 8.2 overviews related work in the area of trail visualization. Section 8.3 introduces the proposed visualization techniques. Section 8.4 presents several visualization results for the analysis of country-scale and world-scale air traffic. Section 8.5 discusses our techniques. Section 8.6 concludes the chapter.

8.2 RELATED WORK

Visual air-traffic analysis techniques and tools can be roughly divided into two classes – decision support systems and exploration systems – as follows.

Decision support systems, such as ATC systems, typically handle low-to-moderate size datasets, such as the region over an airport or city (Fig. 8.1b), or thousands of trails over larger geographical areas. These tools provide sophisticated query mechanisms to support various ATC tasks. The Future ATM Concepts Evaluation Tool (FACET) is capable of quickly generating and analyzing thousands of aircraft trajectories [24]. It provides a simulation environment for the climb, cruise, and descent phases of an aircraft's flight. Traffic patterns are shown in 2D and 3D, under various current and projected conditions for specific airspace regions. Similar systems have been developed by Eurocontrol, the European Organization for the Safety of Air Navigation. For example, the Network Strategic Tool (NEST) [77] is a tool used by air traffic practitioners for airspace structure design and development, capacity planning and post-operations analysis, the organization of traffic flows, the preparation of scenarios for fast time simulations, and ad-hoc studies at local and network level. EPOQUES [94] is a tool which gathers and analyzes radar recordings and audio communications. It proposes underlying techniques to treat Air Traffic Management (ATM) safety occurrences, such as helping operators to detect and analyze situations when two aircraft go beyond safety distance. CoFlight [274] is a flight data processing (FDP) open-architecture framework for the storage, analysis, and visualization of 4D (spatio-temporal) flight data. A comprehensive list of over 50 ATC-related systems and tools is given in [91]. While such systems emphasize the importance of visualization for ATC systems, they all lack high visual scalability and/or the ability to show multiple data attributes at the same time. Specifically, there is no way to continuously navigate between the different levels of abstraction, which makes it harder to link global and local scale patterns.

Exploration systems, in contrast to decision support systems, aim at showing as much traffic data to the user as possible, without prior filtering, so the user can spot unexpected behavior. By next detecting outlier and/or mainstream patterns in such visualizations, users can focus on a subset of the data, and refine their understanding thereof. Many such systems employ a space-filling (also called dense-pixel, or image-based) metaphor [167]: By trying to use each screen pixel to convey data, users can explore larger datasets on a wider range of levels of abstraction, from fine-grained and local patterns to coarse global patterns. Image-based techniques also naturally map to GPU implementations, which helps their computational scalability. For instance, [294] use density maps to show thousands of trajectories of nautical vessels on 2D maps and also to emphasize high-congestion areas. By next combining several density maps, a few attributes can be analyzed simultaneously [225]. [142] use GPU techniques to quickly compute uncluttered layouts of large aircraft trajectories in both 2D and 3D [143]. The FromDaDy system allows interactive linking and brushing of airplane trails to support complex queries in the entire attribute space recorded in the dataset [113]. Density maps are effective to tackle the visual scalability problem, by aggregating spatially close information for trajectory analysis [7, 8, 171]. Multimodal interactions help users in posing complex queries with little effort [149]. Bundling techniques are effective in showing the coarse-scale connectivity structure of a set of trails that link a set of spatial locations in a clutter-free manner [53, 76, 108, 116]. Bundling can also be used to show the dynamics of trails, *e.g.*, how flight patterns change over a geographical area over a week [117]. Our CUBu technique also falls within this class, although the algorithm presented in Chapter 7 does not handle the dynamic aspect. Focus+context interaction techniques help in further reducing clutter and posing complex spatial- and data- queries in trajectory visualizations [114, 139].

8.3 VISUALIZATION TECHNIQUES

We now introduce our image-based visualization techniques for plane trails exploration. Throughout the exposition, we use as running example the one-week French air traffic dataset from [117] (52K flights, about 900K recorded plane positions).

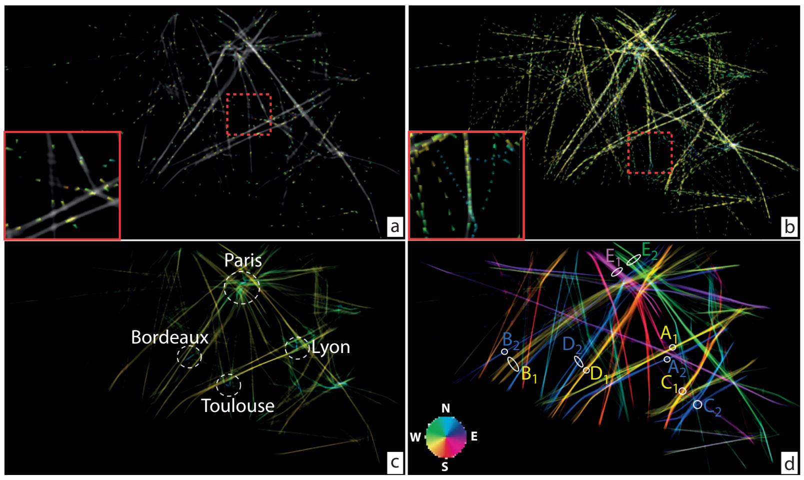


Figure 8.2: Animated multivariate visualization, French airline dataset. (a) Instantaneous plane positions, with color-coded height. (b) Trail segments over short time periods, with color-coded height. Trails over entire studied one-week period with color coding height (c) and direction (d).

8.3.1 *Data model*

We model a flight path, or trail T, as a sequence of points

$$T = \{\mathbf{p}_i = (\mathbf{x} \in \mathbb{R}^2, h \in \mathbb{R}^+, t \in \mathbb{R}^+)_i\} \tag{8.1}$$

which we order along increasing values of t_i. The points $\mathbf{p}_i$ hold recorded samples of the plane's position (x, y), flying altitude h, and possibly additional quantities such as ground speed and air speed. Our dataset is thus a collection $TS = \{T_i\}$. Attributes can be also defined at the trail level, *e.g.*, the flight ID. At an even higher level, we can have attributes at the level of a group of spatially-and-temporally close trails, which we call a trail *bundle* [117].

8.3.2 *Multivariate data shown using animation*

To address the challenge of showing the trail data outlined in Sec. 8.3.1, we first consider using *animation* and *density maps*, akin to [225, 294]. However, we take several different design decisions, leading to different visualizations, as follows.

First, we consider four instantaneous attributes (that is, sampled at all moments t_i, see Eqn. 8.1):

- **A1:** instantaneous positions of in-flight airplanes ($\mathbf{x}$ in Eqn. 8.1);
- **A2:** height along flight trails (h in Eqn. 8.1);
- **A3:** flight directions along trails ($\frac{d\mathbf{x}/dt}{\|d\mathbf{x}/dt\|}$ with $\mathbf{x}$ as given by Eqn.8.1);
- **A4:** airplane flight speed along their flight trails ($\|d\mathbf{x}/dt\|$ with $\mathbf{x}$ as given by Eqn. 8.1).

Given these data attributes, we construct a density map

$$\rho(\mathbf{x}) = \sum_{T_i \in TS} \int_{\mathbf{p} \in T_i} K\left(\frac{\mathbf{x} - \mathbf{p}}{h}\right) \tag{8.2}$$

by convolving the trail-set with a 2D Gaussian or Epanechnikov (parabolic) kernel $K : \mathbb{R}^2 \rightarrow \mathbb{R}^+$ of width h. This step is essentially identical to CUBu's kernel density estimation (Eqn. 7.1).

The density ρ is subsequently interpreted as luminance to become the background of the visualization, similarly to [225]. However, in contrast to [225], we use the density map only as a *context* visualization atop of which our actual fine-grained animation takes place, whereas [225] use the density map as their prime visualization vehicle. Figure 8.2a shows the density map for the French airline dataset. Bright white-gray areas show regions of intense traffic for the entire considered time range. Dark gray regions indicate areas where few or no flights were recorded in this period. Note that this use of the trail density ρ is different from its use in CUBu (Chapter 7): Indeed, CUBu used density simply to alpha-blend trails, and thus essentially show the degree of *overplotting* of trails. In contrast, we show ρ at *all* points $\mathbf{x}$ in our considered 2D domain. This creates a blurred version of the trails, which thus acts as a context information for the actual trail information displayed atop (as explained next).

Next, we have to address the fact that our dataset TS is time-dependent. For this, we consider a so-called sliding time-window $w(t) = [t, t + \Delta]$, which moves with constant speed (given by a user-controlled animation setting) over the considered time range. Here, $\Delta > 0$ controls the size of the time window, and thus the amount of information our animation will show at any moment. Given such a time-window, we select all data points $\mathbf{p}_i \in TS$ for which $t_i \in w(t)$. Rather than drawing entire trails T atop of the background, such as *e.g.* [225] or [117], we now consider *trail segments* $T_\Delta(t)$ which contain all trail sample-points falling in w. We draw these trail segments, textured with a transparency (alpha) texture. This texture is built by placing at the sample point positions $\mathbf{p}_i$ a train of 1D Gaussian half-pulses ϕ_i tangent to the trail segments $(\mathbf{p}_i, \mathbf{p}_{i+1})$. The pulses ϕ_i are scaled so that they are 1 at the location of $\mathbf{p}_i$ and near zero at a distance $\delta\mathbf{v}_i$ downstream the flight path, where $\mathbf{v}_i$ is the instantaneous plane speed at $\mathbf{p}_i$ and $\delta > 0$ is a user-set parameter. The final texture is built by modulating the pulses ϕ_i with a large 1D Gaussian envelope Φ_Δ placed over w and summing up the modulated values. The entire process is explained in Fig. 8.3.

Texturing serves two purposes, as follows. First, setting both Δ and δ to very low values creates images where the arrow-like (high to low alpha) shapes created by ϕ_i, and their motion due to the sliding window $w(t)$, shows the *instantaneous* plane positions at a given time moment (**A1**) as well as their motion along trails (Fig. 8.2a). In contrast, setting δ to low values and Δ to larger values creates 'trains' of arrow-like shapes that slide along trails. Figure 8.2b shows a snapshot from such an animation. Here, short pulses indicate slow-motion planes – indeed, slower planes mean closer-spaced trail sample-points, thus shorter pulses. Analogously, longer pulses show fast planes. Finally, we can add a third attribute to the visualization by using color mapping. For instance, in Fig. 8.2b (inset), we use a blue-to-red (rainbow) colormap to map altitude[2]. We see here a fine-grained blue trail segment indicating a slow, low-altitude, outlier flight in an area with fast (long pulses) and higher (green) flights (**A4**).

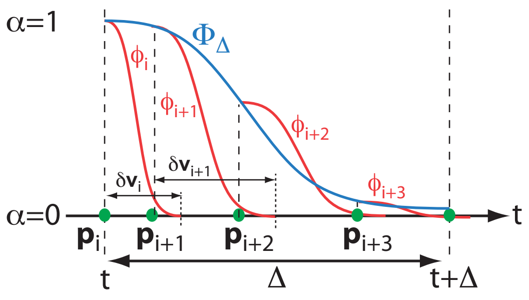


Figure 8.3: Construction of directional pulses for animation.

Increasing both δ and Δ also allows us to smoothly navigate from instantaneous views on the data to more global views. Figure 8.2c shows this for Δ set to roughly 8 hours and δ to 4 hours respectively for our one-week flight dataset. Colors map flight altitude (**A2**). Blue spots indicate regions densely populated by landing zones (airports). Warm lines show in-flight routes. By looking at the latter, we can see that most studied flights have the same altitude. This observation correlates with flight rules for civil aircraft for the studied territory (France). Figure 8.2d shows a similar map, with trails colored now using a directional hue colormap (see colorwheel in the image), thus addressing **A3** over the entire studied time period. Directional

2 As mentioned at other points in this thesis, we are aware of the limitations of rainbow colormaps, but use them here as they prove best for a sparse dataset (trails) being drawn on a bright white background.

color coding lets us discover several close-and-parallel, opposite-direction, flight paths, *e.g.* A_1, A_2; B_1, B_2, C_1, C_2 and D_1, D_2 (going southwest-northeast and conversely); and E_1, E_2 (going roughly northwest to southeast and conversely). Similar patterns (not shown here for conciseness) exist for almost all the other similar-size time intervals in the studied 7-day period. From such images, we can conclude that flights linking pairs of airports follow parallel paths but are structurally not overlapping in space.

However useful in showing the flight directions, flight speed, and overall flight locations, the above visualizations suffer from a certain amount of *clutter*, especially for large values of Δ. Indeed, in such cases, our trail-segment set contains many crossing flights, especially in high-density areas such as close to airports, and if the dataset changes significantly over the studied time period. Understanding flight patterns in such areas is important for many ATC planning tasks [113, 149]. We further help users in getting clearer, less cluttered, insight in such areas by using several transfer functions, as follows:

Alpha transfer function: Consider, for instance, that we are interested in low-altitude flight segments (close to airports). To focus on these regions, we modulate the pulse textures ϕ_i with a nonlinear transfer function $f(h) = \left(\frac{h_{max}-h}{h_{max}}\right)^{k_\alpha}$, where h and h_{max} are the altitude and its maximum value respectively. Values of $k_\alpha < 1$ render low-altitude trail segments gradually transparent, allowing one to focus on the high-altitude ranges. Values of $k_\alpha > 1$ render high-altitude trail segments more transparent, allowing to focus on low-altitude ranges.

Color transfer function: Consider, for instance, that we color map the altitude attribute (h in Eqn. 8.1). If we are interested in focusing on altitude variations for the low-altitude (close to airport) range, we need to dedicate more dynamic range to this signal range. To do this, we apply a transfer function $f(x) = x^{k_{color}}$ to the normalized altitude attribute prior to color mapping. Values of $k_{color} < 1$ emphasize high altitude ranges. Values $k_{color} > 1$, in contrast, emphasize low altitude ranges.

Figure 8.4 shows the effects for our French airline dataset. Image (a) shows the effect of $k_{color} = 1$ and $k_\alpha = 2$. As the high-altitude trail segments become more transparent, we can now better focus on the airport zones and thus the landing and take-off trail segments. These are apparent on the image as colder colors (blue). Image (b), taken for a longer time-window Δ value, shows the effect of $k_{color} = 0.5$ and $k_\alpha = 1.5$. We see now more and longer trails, since Δ is longer. However, the clutter due to overdraw stays limited, due to the fading out of high-altitude trail segments caused by k_α. The low k_{color} value allows us to visually separate the warm-colored cruise trail segments (which have higher altitude) from the cold-colored landing and takeoff segments (which have lower altitude). Images (c-e) show three snapshots from our one-week period taken at dif-

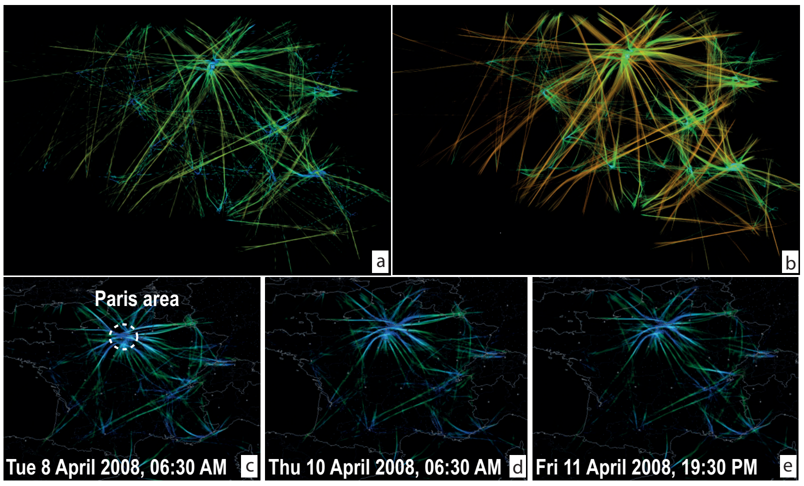

a
b
Paris area
Tue 8 April 2008, 06:30 AM
c
Thu 10 April 2008, 06:30 AM
d
Fri 11 April 2008, 19:30 PM
e

Figure 8.4: Emphasizing specific flight ranges and decreasing occlusion by color and alpha transfer functions.

ferent moments of the morning and evening, for $\Delta = 30$ minutes. Here, by using $k_\alpha = 3$, we are able to declutter even more of the crowded airport regions, and see the so-called 'approach lanes' of the planes, *i.e.* the general paths that planes take when taking off or landing at an airport. Although images (c-e) are for three different days and two diffent times of day, we notice that the approach lanes above the Paris area are quite similar. This is not a trivial finding since, if we look at other times of day, such patterns are quite different. The found explanation (in discussions with ATC controllers) is that planes that land and/or take off early in the morning or late in the evening are typically long-distance hauls, which have more stable approach lanes than shorter-range flights common during the day.

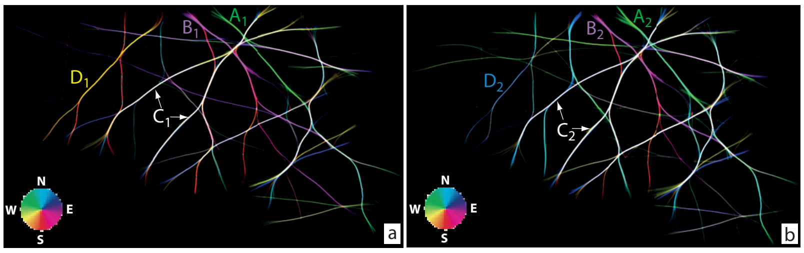


Figure 8.5: Emphasizing airport connection patterns by trail bundling.

8.3.3 *Bundling-based simplification*

As shown so far, our flight visualization offers several scales, or levels of detail, at which the data can be examined – ranging from instantaneous plane positions to trail fragments and ending with large trail sets over several days. However, apart from this *temporal* multiscale, we can also exploit the *spatial* multiscale of our trail data. Looking at *e.g.* Fig. 8.2d, we see that trails come naturally grouped in sets of closely spaced, relatively parallel, trails. This observation has been exploited by many bundling algorithms that simplify the visualization by bringing together all trails in such sets, *e.g.* [53, 76, 108, 116]. The resulting images, although they distort the spatial information, are much more effective than trails in showing the connectivity patterns between airports, and how these change in time. Similar examples of bundling of static datasets donw by our CUBu method are shown in Chapter 7.

Recently, Hurter *et al.* [117] have shown how trail bundling can be applied to airline trails, by applying the efficient and robust KDEEB bundling algorithm [116] to a so-called 'streaming graph' containing only trails whose start time moment falls within a sliding time-window. However, their solution does not show any additional attributes atop of the emerging bundles, such as flight directions, height, or speed. Moreover, as discussed in Chapter 7, KDEEB is one up to two orders of magnitude slower than CUBu.

We extend here the idea of dynamic bundling from [117] by combining our fast CUBu bundling with our multivariate attribute-based animations presented earlier in Sec. 8.3.2. In detail, we apply CUBu to trails selected by our time-window $w(t)$. This delivers a set of bundled trails. Next, we map on these bundled trails the attribute values of the corresponding sampling points (for identical time moments) from the original, unbundled, trails. In the end, we use the visualizations described in Sec. 8.3.2 to create the final images.

Figure 8.5 illustrates this idea. Images (a,b) use the same color coding as in Fig. 8.2d. However, the trails are now given by two frames of the *bundled* flight graph, which correspond to two moments in two different days in our one-week dataset. Since trails are bundled, geographical (spatial) information is lost: Bundles indicate now just *connections* between airports, rather than actual flight paths. Still, directional color-coding is useful to show temporal insights. First, we see that the connection pattern is roughly identical for the two studied moments. Flights in bundles A and B keep their directions over time, respectively northwest (green) and southeast (pink). Flights in the big central white bundle structure C go equally in both directions at both studied moments, since white is the result of additively blending opposite colors in our colormap. In contrast, flights in bundle D go southwest (yellow D_1 in Fig. 8.2e) and then return northeast at moment 2 (blue D_2 in Fig. 8.2f). All the other visualizations described in Sec. 8.3.2, such as animating pulses along bundles to show flight directions, or using transfer functions to focus on specific data ranges, are further directly applicable on the bundled trail-set.

8.3.4 *Congestion detection*

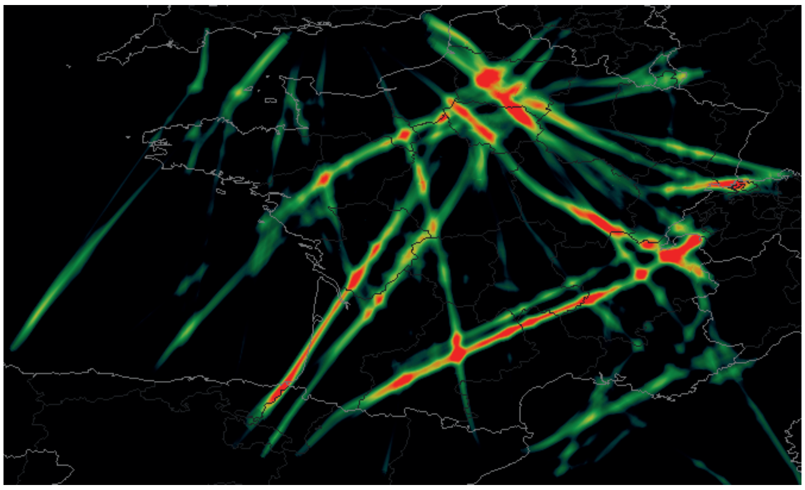


Figure 8.6: Congestion detection. The kernel size corresponds to a time-interval of 30 minutes. Alpha blending is used to focus on higher flights.

An important and frequently occurring task in movement data analysis is detecting and examining so-called *congestion areas*, *i.e.* spatial zones where many vehicles are present at a given time moment [113, 225]. In ATC, such areas are particularly important to prevent air traffic congestion and, thus, delays or an increase in fuel consumption. On small spatial scales, congestion areas become collision areas, *i.e.* zones where a high risk of vehicle collision exists. Correlating the appearance of such zones with other parameters can give important insights in the reasons why such problems occurred and ways to solve them [24, 77, 94].

An early, and relatively simple, approach to congestion area detection was given by [225] for vessel (ship) trails: By visualizing the density map ρ (Eqn. 8.2), we can detect zones of high vehicle densities. However, this solution was proposed in a *static* setting: There is a single density map ρ computed for the entire studied time period (or alternatively put, for the entire trail-set TS). As such, *dynamic* congestion patterns that occur and disappear on smaller time-scales are not visible. Secondly, this basic solution does not assume there is a higher probability of collision in the direction of vehicle motion and for rapid vehicles than for other situations. Such situations are more relevant for our use-case, where we consider airplanes, which more more rapidly than ships.

We extend this idea by using anisotropic kernels K in Eqn. 8.2. In contrast to the isotropic radial kernel, such kernels are larger in the direction of instantaneous motion of a vehicle than in other directions. A simple way to implement this is to use *e.g.* elliptic kernels whose large axis is tangent to the trail, *i.e.*, oriented along the normalized vector $\frac{d\mathbf{x}/dt}{\|d\mathbf{x}/dt\|}$, and scaled to be equal to the instantaneous velocity. A further refinement involves using asymmetric kernels, which are longer in the motion direction than in the opposite direction, thereby modeling the fact that congestion or collision is more probable in *front* of a moving vehicle than behind it. Other kernels can be immediately used to model other types of congestion probabilities, as and when desired. Moreover, the computation of the density map ρ still follows Eqn. 8.2, using the suitable kernel K

Figure 8.6 shows the result of visualizing this congestion density map for the French airline dataset. Here, we color mapped the quantity $\max(\rho-1, 0)$ to a rainbow colormap. Indeed, ρ is by construction equal or larger than 1 at every plane location, and only values larger than 1 indicate a congestion probability, *i.e.*, the overlap of two kernels corresponding to two different planes that are close to each other. We also used $k_\alpha = 0.2$ to focus on higher-altitude trail segments, as we are more interested to detect and assess in-flight congestion rather than congestion close to or on the airstrips. The kernel size h was set to be equivalent to a duration of 30 minutes, thereby modeling a use-case where if several planes at high altitude get closer to each other than a flight time of 30 minutes, we consider the area as being congested. The red patterns visible in the image delineate quite clearly the emerging congestion patterns. These patterns are not (easily) visible using any of the earlier-presented visualizations. We notice that the congestion areas are, in most cases, well aligned with the the main flight routes, which is expected. However, we also see a few red blobs which do not follow the elongated structure parallel to these routes. These indicate congestion areas that occur at the *intersection* of several routes rather than on a single route.

8.4 ANALYSIS RESULTS

We used our visualizations to analyze several trail datasets over different space and time periods. Statistics for the datasets shown in this chapter are given in Tab. 3. Besides the French dataset, we show also a dataset with three days of flights over Europe and one with one-month flights over the entire world. Our goal was the *explorative* scenario outlined in Sec. 8.2, which consists of two related questions:

- Given a large and unknown dataset, can we (as users) quickly form a general impression on the distribution of flights in terms of spatial location, direction, speed, and height?
- Can we discover outlier flight patterns, which diverge, in some significant way, from the overall flight patterns in the same dataset?

We next present several of our findings that we obtained when trying to answer the above questions.

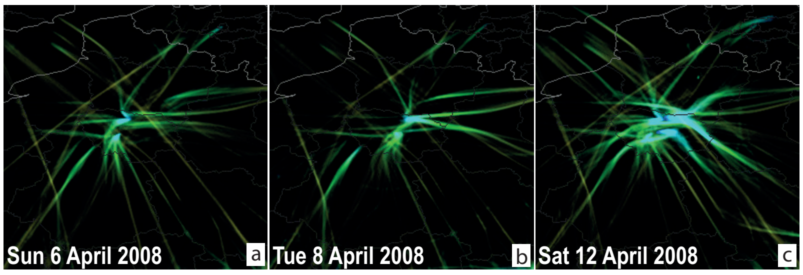


Figure 8.7: Height-colored trails over a duration of 24 hours with an alpha-based emphasis on low flights (and airports). We see a clear difference in landing directions Sunday *vs* Tuesday. Saturday shows a significant increase in traffic around Paris.

Attributes	French air-traffic	Europe air-traffic	World air-traffic
start date	06/04/2008	01/06/2013	01/06/2013
end date	12/04/2008	03/06/2013	30/06/2013
# flights	52547	50984	748057
# sample points	870880	873240	14711646

Table 3: Dataset statistics for examples in this chapter.

Outlier landing/takeoff patterns: In Fig. 8.4 (d-e), we found that landing/takeoff patterns over the Paris area, for three moments, are quite similar. However, we cannot generalize to infer that such patterns are constant for *all* moments. Also, the zoom level in Fig. 8.4 is too low, so potential small-scale pattern changes would not be visible.

We repeated the experiment shown in Fig. 8.4 (d-e) at a finer zoom level. Also, we set Δ to 24 hours, to show more data in one animation frame, thereby allowing us to move the animation faster to cover a longer time period quicker. Next, we watched the animation for our one-week dataset. Pattern changes are easily spotted as changes in the animation. We thus discovered that pattern changes indeed exist. Figure 8.7 shows three frames from this animation, for three different days. We quickly see that the Saturday traffic is much more intense than on Sunday and Tuesday. This confirms the expected week variation of flight patterns. More interestingly, the Tuesday landing/takeoff routes are quite different than the ones for the other two days. To explain this further, we looked up data for wind direction around the Paris area for these three days, and found out that the wind patterns on Tuesday were quite different than for the other two days. This explains our finding, as ATC rules indicate that landing/takeoff flight segments are

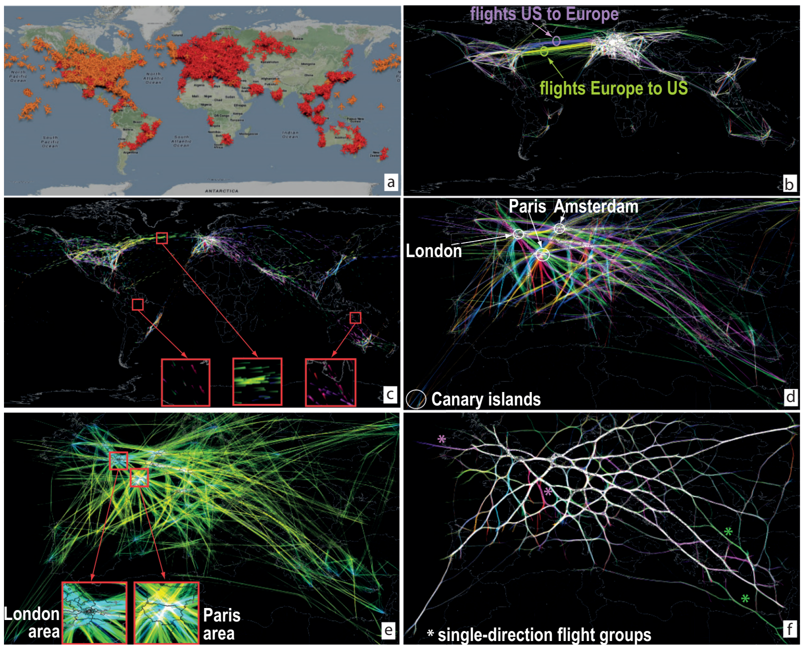


Figure 8.8: (a,c) Overview of world traffic, June 1, 2013. (d-f) Details over Europe (see Sec. 8.4).

indeed computed based on wind directions.

Global flight patterns: We now consider a larger dataset, covering the entire world. The data, available online [207], is gathered continuously by hobbyists that record ADS-B plane feeds [1] used by commercial and private planes to transmit their name, position, altitude, callsign, status, and other information, and consolidated into a global server. ADS-B is gradually replacing radar as the most efficient method for ATC, so our visualizations will potentially become directly relevant for ATC-related tasks in the near future. In contrast to the French airline dataset, obtained directly from the French ATC authorities, the world dataset is far less uniformly sampled, depending on the position of hobbyist receivers throughout the world. However, this dataset is orders of magnitudes larger (see Tab. 3). We processed this data to create the trails dataset necessary for our visualization, by matching IDs of the same flight, removing duplicate sample points (coming from different beacons), and separating flights having the same ID that occur during different days.

Figure 8.8 shows an overview of the world traffic on June 1, 2013. Image (a) is a snapshot from [207], showing plane *positions* at one moment during the day with icons. Besides flight densities, little is visible on this image. Image (b) shows our visualization of flight *routes* for that day, color-coded by flight direction. As for the smaller French dataset (Fig. 8.2d), we see here too that flights linking the same (close) airports but having different directions follow parallel, but separated, lanes – such as the broad one between Europe and the US. However, the densely flown regions, such as Europe, are too cluttered at this scale. One solution to de-clutter is to reduce the parameters δ and Δ, to focus on shorter time-ranges. Image (c) shows this result. Here, the arrow-like glyphs become visible and as such indicate the flight directions more clearly (see insets). As such, the European region also becomes more de-cluttered. To further de-clutter and obtain local detail, we zoom in over Europe (image (d)), and increase back δ and Δ to see full one-day trails, like in image (b). We can again see here the lane separation patterns, such as the one linking the Canary islands with the mainland and connecting the main hubs, *e.g.* London, Paris, and Amsterdam with the rest of the map. Image (e) shows the same region, this time color-coded by altitude. Low-flight zones such as airport areas are blue, and cruise segments are green. We see that the average cruise heights over Europe are quite similar. The sizes of the blue spots indicate the extent of low-flight zones close to airports. Interestingly, the entire of south-east Britain is such a zone, which is not crossed by any high-altitude flight (yellow trails). In contrast, the Paris area shows a similarly-sized blue zone, but which gets crossed by quite many high-altitude flights.

Image (f) shows the Europe traffic with trail bundling, colored by flight directions. We notice here many white bundles: These are parallel *and* close trails which have nearly equal counts of flights in opposite directions. Indeed, since the KDEEB algorithm works by grouping trails in distance order, trails that end up in the same bundle are by construction the closest ones to that bundle's location. And, secondly, since trail colors are additively blended *and* we use directional hue-coding, we achieve gray (or white) when a bundle contains (nearly) equal trail amounts running in opposite directions. We can thus infer that most trail groups over Europe over the considered day contain roughly equal numbers of flights in opposite directions. This situation was different for the two considered day *moments* for the *French* airspace shown in Fig. 8.5. Thus, we infer that, at a coarser day-over-Europe scale, air traffic is more balanced. Finally, we see in Fig. 8.8f also a few outlier colored trails (see markers in image). These are groups of flights that go in a *single* direction, *i.e.*, there are no opposite-direction flights in the same spatial region for the entire considered day.

However useful, the above images do have an important limitation: They group trails that represent airplanes flying in opposite directions. As explained in Chapter 7, this type of simplification can be too high for use-cases where we aim to answer questions about *directed* connections between various positions on a spatial map. A separate technical problem regards color mapping: When undirected bundling is used, trails being close to each other and going in opposite directions get bundled together. In turn, when using color-mapping to show direction, and when using a variable-hue colormap (such as most directional colormaps are), colors that represent different direction values get blended together, resulting in wrong insights. This is a well-known problem that can be easily explained by the fact that blending (summing) and color mapping (translating data values into colors) are not commutative operations for variable-hue colormaps (see [271], Chapter 5, for more details). Using a blending safe' colormap, such as the grayscale colormap or other single-hue colormaps, is not a good option, as such colormaps are not well suited for encoding directions.

As such, we propose to address the above issues using directional bundling, which has been discussed in Chapter 7. The key advantage hereof is that close trails that run in opposite (or, more generally, highly different) directions are not aggregated in the same bundle. This allows both the creation of images where one can reason about directional connections in the dataset, and also solves the technical problem of blending colors from a variable-hue colormap.

To achieve directional bundling, we cannot use existing directed bundling methods [108, 206, 231], as these are orders of magnitude too slow for the real-time requirements of dynamic trail exploration, where we need to create tens of frames (thus, bundled layouts) per second to yield a smooth animation. Note that the required speed here is much higher than when generating static directionally bundled images, such as covered by the examples in Chapter 7.

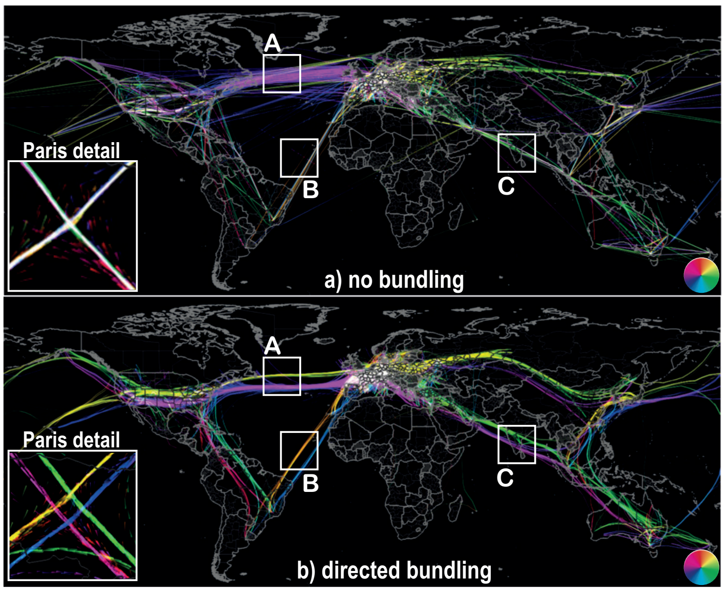


Figure 8.9: World flights (June 1st 2013), raw *vs* directed bundling (Sec. 8.4).

CUBu's fast directional bundling solves the speed problem. Figure 8.9 shows a frame of the dynamic bundling of all world flights in the database [207], corresponding to the morning of June 1st, 2013 (about 26K flights from a total of 750K flights in the database). The unbundled flight display (Fig. 8.9a) shows several clutter areas, where we see only a single direction-color (*A*) or false colors, not even existing in our directional colormap (B, C). The fact that undirected bundling creates false colors is more visible in the detail inset in Fig. 8.9, corresponding to a small zone above Paris: Here, we see an X-like crossing pattern colored white. However, the directional colormap we use does not contain white (see color wheel legend, bottom-left in the images). Directional bundling (Fig. 8.9b) clearly separates trail-sets running in opposite directions. We now see that in all regions *A..C* there is symmetric traffic in both directions. This tells us how planes connect different regions of the world – for example, for flights linking Europe with North America (detail A), we see that flights from Europe to North America (Fig. 8.9b, purple bundle) fly at a lower latitude than flights from North America to Europe (Fig. 8.9b, yelow bundle). Quite interestingly, the same pattern can be seen for most other long-haul flights over the entire world – flights going roughly to the east are at lower altitudes than flights going to the west. Also, no false colors are created when directional bundling is used (see inset in Fig. 8.9b).

8.5 DISCUSSION

Several aspects are relevant to our presented techniques, as follows.

Scalability: We implemented our visualization in Python and C++, using OpenGL pixel shaders for the rendering part (texture computation, blending, transfer functions, and congestion map, see Sec. 8.3). For bundling (Sec. 8.3.3), we use the CUBu method described in Chapter 7. Table 4 shows our timings on a 2.6 GHz Windows PC with a NVidia 690 GTX card, for various trail selections. The bundling cost is roughly linear with the number of sample points. Compared to the earlier fastest-known bundling technique to us (KDEEB, [116]), our bundling is about *30 times* faster, on identical hardware. These results are in line with those presented in Chapter 7 for static bundling. The slightly lower speed-up shown here as compared to Chapter 7 can be easily explained by the implementation overhead required to handle dynamic bundling, *i.e.*, selecting trails that fall in the sliding time window, passing these to the GPU, getting the bundled results from the GPU, and displaying these using OpenGL CPU-side calls. The computational complexity of our technique is linear in the number of trail sample points falling in the considered time-window of length Δ. Given the above-mentioned design decisions, we can all in all create real-time animations of flight data for a few million sample points. This performance was not achievable with earlier techniques [77, 113, 117, 139].

Limitations: While our technique has significantly less visual clutter than *e.g.* [113, 117, 139] by means of transfer functions and bundling, highly dense flight areas viewed at a coarse scale will still have a high amount of overlapping flights. This problem is solved in [225] by showing only aggregated information. In contrast, we choose to tolerate a clutter to be able to show individual outliers in such areas. To increase resolution, we use a

Statistics	# flights	# sample points	bundling time (msecs)
Dataset 1	50984	683216	74
Dataset 2	23433	886323	89
Dataset 3	50984	1280680	124

Table 4: Dynamic CUBu bundling statistics.

large 60-inch touchscreen, which makes finer-grained patterns easier visible. A second limitation concerns the number of attributes that we can show simultaneously on a trail – currently, this is limited to three (speed, direction, and altitude). Showing more attributes is an open challenge to all similar research. A final, implementation-level, limitation concerns the architecture of our dynamic bundling solution. As outlined above, bundling is done by CUBu fully on the GPU, but the final rendering and interaction is done by issuing OpenGL calls from the CPU. As such, at every frame, we need to transfer data from the GPU to the CPU, a process which is well-known to be slow. This can be easily accelerated by shading the bundle data between CUDA and OpenGL, by using so-called vertex buffer objects (VBO's), at the expense of a slightly more complex implementation. Early results in this direction (not included in this thesis due to their preliminary status and insufficient benchmarking) indicate that one can remove more than half of the rendering overhead by this approach.

8.6 CONCLUSIONS

We have presented a set of visualization techniques for the interactive exploration of very large movement datasets emerging from Air Traffic Control (ATC). On the application side, our main goals were to achieve high information density with limited clutter, present several movement attributes such as altitude, position, and speed at the same time. On the technical side, our goals were to extend CUBu, our fast bundling technique presented in Chapter 7, to handle multivariate and time-dependent trail-sets, and to demonstrate its applicability for assisting the above-mentioned application goals.

We achieve the above goals by following an image-based visualization design based on density maps (to show amount of flights), animation (to show direction and change in flight patterns over time), and graph bundling (to show coarse-scale similar patterns and their change over time). We achieve computational scalability by using our CUBu technique presented in Chapter 7, which can be easily adapted to handle dynamic trail-sets by using a sliding-window technique. The visualization design and implementation also allows users to smoothly navigate, in both space and time, between local detail and global patterns. We demonstrated our techniques on several flight datasets ranging from hours over a single country to one month over the entire world.

Although visual scalability is still challenged by the sheer amount of information to be shown, our method is considerably more scalable both in visual space and computational complexity than current methods used for the same types of datasets and analyses, most notably the one of Hurter *et al.* [117] In the future, we plan to augment our visualization by adding interactive queries in order to enable users to compare and search spatio-temporal patterns of interest, and also enhance the image-based design to allow for the display of more data attributes at the same time.

ACKNOWLEDGEMENT

We would like to thank the maintainers of `planefinder` [207] for letting us use their plane-tracking database. Without having access to this worldwide database of airplane trajectories, large-scale validation of the visualization methods proposed by us in this chapter would not have been possible. Separately, we thank Christiaan Arnoldus, from the Institute Johann Bernoulli, University of Groningen, the Netherlands, for his help with the implementation of various parts of the dynamic CUBu algorithm's infrastructure.

9 CONCLUSION

We conclude here our work on visual analytics of multidimensional time-dependent trails with a revisit of the core research question introduced in Chapter 1, and a reflection on the extent, ways, and limits of our proposed answers.

The main research question (*How can visual analytics help understanding time-dependent multidimensional data to support the analysis of behavior of 3D shapes?*) has been, first, refined by the introduction of the key concept of dynamic *trails*. These trails are simple (2D or 3D) curves which characterize the motion of shapes (or shape parts). As such, our work's common denominator is the understanding of such trail-sets by visual analytics methods.

To manage the domain of interest, we further split the research question based on the scale and type of dynamic 3D shapes we considered into a small-scale problem and a large-scale problem, as described next. To better allow the comparison of the various challenges, design decisions, and results obtained in each case, we discuss the two sub-domains of our work along the following axes:

1. *level-of-detail:* This dimension describes the amount of information available in each of the dynamic shapes being part of a studied dataset;
2. *size:* This dimensions describes the amount of shapes being part of a single studied dataset, as well as the amount of samples along the trail of a single shape;
3. *dimensionality:* This dimension describes the amount of independent variables being recorded for each dynamic shape. At minimum, this is equal to a time-dependent position in 2D or 3D;
4. *data acquisition:* This dimension describes the procedure involved in acquiring, or preprocessing, the input data in order to obtain the trail-sets we want to study;
5. *visual analytics:* This dimension describes the type of techniques used to visually explore the data in order to obtain the desired understanding;
6. *validation:* This dimension describes the goal and procedure used to validate the findings obtained by the proposed visual analytics methods.

We next describe our results obtained in the analysis of the dynamics of trail-sets from small-scale, respectively large-scale, trail-sets along the above-mentioned dimensions.

9.0.1 *Small-scale trail-sets*

In this context, we consider the analysis of a small-scale shape – the dynamics of a cow udder. The involved use-case concerns the design of a computer vision front-end for the construction of automati milking robots (AMDs) in the dairy industry. The problem is characterized by the following aspects:

1. *level-of-detail:* The considered shapes are quite complex (deformable, having a wide variability in form and size). As such, even if we are interested chiefly in the dynamics of just four points (the teat tips), studying the acquisition of the dynamic data is unavoidable;

2. *size:* The size of this dataset is small in number of shapes – we track three, maximally four, teats per udder. However, in terms of samples per trail, such datasets an be relatively large, as a trail can contain thousands of position samples;

3. *dimensionality:* The raw trail-set has a low dimensionality – for each trail-point, we only have its 3D position and timestamp. However, we need to study also the various variables produced by the tracking system that delivers us suh trail-sets from the raw input video information (see below). As such, the dimensionality of a trail point can be between 10 and 15;

4. *data acquisition:* The available input data, though dynamic (time-of-flight videos), does not deliver us directly the desired trail-sets. As such trail-sets are not readily available by existing means, and since their generation from the available video data is a practical problem of interest to the main sponsor of this work (Lely Technologies), we dedicate special attention to the design of computer vision algorithms for trail extraction. Chapter 5 presents a family of such algorithms, based on various technical elements, the most important being our proposed 2D and 3D template-based teat detectors and deformable model tracking. Separately, we present a method to segment grayscale and color images using an extension of the novel concept of dense skeletons (Chapter 4). While this method, unfortunately, has not proven useful for supporting our teat tracking application, it has given very good results in the context of segmenting high-resolution general-purpose images. The main result of this work is that we showed that it is possible to accurately track teat configurations from low resolution, noisy, variable-viewpoint video sequences which contain large amounts of camera and subject motion as well as occlusions. To our knowledge, our results surpass the state-of-the-art in computer vision based tracking results for AMD devices;

5. *visual analytics:* The key use of visual analytics in this context is to study the performance of our proposed tracker, so as to understand the causes of its limitations, and, when possible, to correct these

(Chapter 6). The employed visualization techniques we propose here are relatively simple – timelines, linked views, and multidimensional projections. However, the key added value here is that we showed how such relatively simple techniques can open the 'black box' of tracker design, and reveal subtle constraints and limitations of such systems to the end user. To our knowledge, this is a novel utilization of visual analytics, not used in the context of validating and/or improving computer vision trackers;

6. *validation:* As we lack ground truth information for our trail-sets, we perform validation by using a proxy ground truth, computed by brute-force fitting of a 3D deformable udder model to the depth information in the input videos. Comparing these results with those delivered by our tracker, we find a very good match. All in all, we conclude that our proposed tracker can successfully handle over 90% of the frames available in our test videos. As outlined earlier, the key merit of visual analytics here was to help us narrow down the search space in the large design and parameter space, and offer us efficient ways to assess and improve our tracker's performance.

9.0.2 *Large-scale trail-sets*

Our second application context concerns the analysis of several large-scale trail-sets, such as the trajectories of airplanes over countries, continents, or even the entire world; eye movements recorded from subjects performing a task; relations in a node-link graph drawing; and streamlines describing the motion of 3D fluid flow. These problems are characterized by the following aspects:

1. *level-of-detail:* The considered shapes in this context are relatively simple – in all cases, they can be seen as point-like objects moving in 2D or 3D. This contrasts the high complexity of the deformable udder shapes discussed earlier in Sec. 9.0.1;

2. *size:* Similarly to the above, and in contrast to our small-scale shape context, we consider now large to very large sets of trails, varying between tens of trails (eye tracks) to thousands (streamlines), and tens up to hundreds of thousands (planes and node-link relations). Such sizes pose very different problems in terms of computational and visual scalability as compared to the small-scale context discussed earlier;

3. *dimensionality:* Our trails have a relatively low dimensionality – 2D or 3D point positions, time stamps, trail IDs, and occasionally additional information such as flight height and speed for the airline datasets considered. All in all, the dimensionality of such datasets is not more challenging than that of the trail-sets discussed in the earlier small-scale context;

4. *data acquisition:* In contrast to the situation we had for our small-scale trails, data acquisition is not an issue for our large-scale context: We avail here directly of the required trail-sets, which are delivered directly by third-parties (ATC operators, flight databases, or eye-tracking experiments), easily computed by standard methods (streamline tracers), or directly available from the application context (link positions in a graph layout). As such, all the problems related to extracting reliable trail-sets, and validating the extraction procedure, which we discussed in the context of small-scale trail-sets (udder tracking) are not applicable here;

5. *visual analytics:* Since we do not have the problem of fine-tuning or validating a trail extraction procedure, visual analytics focuses now of very different tasks than those described for the small-scale context. Specifically, since we now have very large datasets, the key goal is to obtain a *simplified* visualization that highlight the essential patterns present in the data, and also reduce the clutter caused by drawing too many trails. Our key contribution here is the proposal of CUBu, a method to generate bundled visualizations of trail-sets (Chapter 7). We show how CUBu can be efficiently implemented on the GPU, being able to create high-quality bundled drawings of trail-sets up to a million trails in under a second on consumer grade GPUs. We also show how CUBu can emulate the bundling styles of most existing comparable methods by parameterizing its degrees of freedom – thereby justifying our dubbing of CUBu as an 'universal' bundling method. Separately, in Chapter 7, we introduce a method for producing a continuous interpolation of different levels-of-details of trail visualizations, demonstrate it for the depiction of 3D vector fields, and discuss it in relation with the natural multiscale generated by CUBu's bundling. In Chapter 8, we extend CUBu to handle time-dependent multivariate trails, and demonstrate this extension for the analysis of dynamic trail-sets of airplanes in an ATC context;

6. *validation:* In contrast to the use-case described for small-scale trail-sets, we do not have now the issue (or goal) of validating a set of trails with respect to a ground truth, since our trails *are* the actual ground truth. For large-scale trail-sets, validation concerns other aspects: Specifically, we validate the technical quality of CUBu by comparing it both in terms of results and computational scalability with state-of-the-art bundling methods. This comparison shows that CUBu delivers similar quality, but is up to 100 times faster than the fastest general-trail-set bundling method in existence at the time of writing our results. Application-wise, we validate our bundling results by showing that they deliver comparable insights to other methods (see the case of simplified 3D flow visualization in Chapter 7), or by showing their concrete added-value in supporting specific use cases (see the ATC scenarios in Chapter 8).

9.0.3 *Impact*

Our work presented in this thesis has led to several notable follow-ups. On a practical side, our work in designing the teat-tracking solutions in Chapter 5 has helped an actual company (Lely Technologies) in their work for the design of the next generation of AMD machines. On a research side, our CUBu trail bundling algorithm has been the basis of numerous extensions in the field by other researchers, such as the visualization of errors in multidimensional projections [168, 169]; the visualization of the behavior of deep neural networks [212]; and the dynamic visualization of software evolution [245]. The performance of CUBu has been exceeded only very recently, and marginally (by a factor of roughly 3), using a refinement of our design [153]. Separately, our work on dense skeletons for image segmentation and analysis (Chapter 4) has formed the basis of a new method for image compression that achieved better results, for similar quality, than the well-known JPEG compression [273]. Our work on simplified visualization of streamlines (Sec. 7.6) has led to recent developments in the visualization of various types of 3D vector fields [121]. On a less scientific level, CUBu has been featured on the cover image of one of the top recent textbooks in data visualization [271].

9.0.4 *Future work*

Many directions of future work are possible based on the material presented in this thesis. Without claiming exhaustivity, we believe the following to be the most promising, and lowest hanging, fruits:

First and foremost, the flexibility of CUBu makes it nearly directly possible to explore the visual simplification of other kinds of trail-like data. The early experiments in simplifying streamline-based visualizations of 3D fluid flow in Chapter 7 deserve detailed attention to be generalized for a true volumetric simplification of 3D fluid flow visualizations, possibly extending the even-streamline designs in [129, 155, 174, 281]. Separately, a similar extension to CUBu could be readily envisaged for the simplified visualization of DTI fiber tracts, along earlier results in [27, 78, 121, 271].

A more challenging extension regards the simplified visualization of multivariate trails using bundling. The key difficulty here is how to aggregate multiple attribute values, recorded at multiple sample points, which end up being displayed over the same pixel in a bundled visualization. This challenge is fundamental to the bundling metaphor, as it generalizes the challenge of overdraw which is present in all (even non-atributed) bundled drawings. CUBu offers the computational and implementational basis for this extension, given its generic design and its very high computational speed.

Finally, our results shown in Chapter 6 support the claim that visual analytics is a powerful, and easy to use, instrument for getting insights in complex algorithms such as computer vision tracker pipelines. Generalizing this idea, it is definitely worth exploring how, and how much, visual analytics can indeed 'open up the black box of machine learning methods' [36]. Recent results, which not coincidentally use some of the results in this thesis (CUBu) show that this is indeed possible and effective [212]. It is up to future research to (re)define and refine such promising results.

BIBLIOGRAPHY

[1] ADS-B. Automatic dependent surveillance broadcast, 2014. `www.ads-b.com`.

[2] A. Agarwal and B. Triggs. Recovering 3D human pose from monocular images. *IEEE TPAMI*, 28(1):44–58, 2006.

[3] N. Ahuja and J. Chuang. Shape representation using a generalized potential field model. *IEEE TPAMI*, 19(2):169–176, 1997.

[4] A. Ali and J.K. Aggarwal. Segmentation and recognition of continuous human activity. In *Detection and Recognition of Events in Video, 2001. Proceedings. IEEE Workshop on*, pages 28–35, 2001. DOI 10.1109/EVENT.2001.938863.

[5] N. Amenta, S. Choi, and R. Kolluri. The power crust. In *Proc. ACM Symp. on Solid Modeling and Applications (SMA)*, pages 249–266, 2001.

[6] G. Andrienko, N. Andrienko, and S. Wrobel. Visual analytics tools for analysis of movement data. *ACM SIGKDD Exploration Newsletter*, 9(2):38–46, 2007.

[7] G. Andrienko, N. Andrienko, M. Burch, and D. Weiskopf. Visual analytics methodology for eye movement studies. *IEEE TVCG*, 18(12): 2889–2898, 2012. ISSN 1077-2626.

[8] G. Andrienko, N. Andrienko, and M. Heurich. An event-based conceptual model for context-aware movement analysis. *Int. J. Geogr. Inf. Sci.*, 25(9):1347–1370, September 2011. ISSN 1365-8816.

[9] A. Appel, F. J. Rohlf, and A. J. Stein. The Haloed Line Effect for Hidden Line Elimination. *ACM SIGGRAPH Computer Graphics*, 13(3):151–157, August 1979. DOI http://doi.acm.org/10.1145/800249.807437.

[10] P. Arbelaez, M. Maire, C. Fowlkes, and J. Malik. From contours to regions: An empirical evaluation. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 2294–2301. IEEE, 2009.

[11] O. Arif, W. Daley, P. Vela, J. Teizer, and J. Stewart. Visual tracking and segmentation using time-of-flight sensor. In *Image Processing (ICIP), 2010 17th IEEE International Conference on*, pages 2241–2244. IEEE, 2010.

[12] D. Auber, Y. Chiricota, F. Jourdan, and G. Melan con. Multiscale visualization of small world networks. In *Proc. IEEE Infovis*, pages 75–81, 2003.

[13] M. Aupetit. Visualizing distortions and recovering topology in continuous projection techniques. *Neurocomputing*, 10(7-9):1304–1330, 2007.

[14] S. Avidan. Support vector tracking. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, volume 1, pages I–184–I–191 vol.1, 2001. DOI 10.1109/CVPR.2001.990474.

[15] X. Bai and L. Latecki. Path similarity skeleton graph matching. *IEEE TPAMI*, 30(7):1282–1292, 2008.

[16] D. H. Ballard and C. M. Brown. *Computer Vision*. Prentice Hall Professional Technical Reference, 1st edition, 1982. ISBN 0131653164.

[17] Y. Bar-Shalom and T. Fortmann. *Tracking and Data Association*. Mathematics in Science and Engineering Series. Academic Press, 1988. ISBN 9780120797608.

[18] D. I. Barnea and H. F. Silverman. A class of algorithms for fast digital image registration. *IEEE Transactions on Computers*, C-21(2):179–186, Feb 1972. DOI 10.1109/TC.1972.5008923.

[19] H. Baya, A. Essa, T. Tuytelaars, and L. V. Gool. Speeded up robust features. *CVIU*, 110(3):346–359, 2008.

[20] R. Becker, W. Cleveland, and M. Shyu. The visual design and control of trellis display. *Journal of Computational and Graphical Statistics*, 5 (2):123–155, 1996.

[21] M. Berger, A. Tagliasacchi, L. M. Seversky, P. Alliez, G. Guennebaud, J. A. Levine, A. Sharf, and C. T. Silva. A survey of surface reconstruction from point clouds. *CGF*, 2016. DOI 10.1111/cgf.12802.

[22] E. Bier, M. Stone, and K. Pier. Enhanced Illustration Using Magic Lens Filters. *IEEE Computer Graphics and Applications*, 17(6):62–70, November/December 1997. DOI http://doi.ieeecomputersociety.org/10.1109/38.626971.

[23] E. A. Bier, M. C. Stone, K. Pier, W. Buxton, and T. D. DeRose. Toolglass and Magic Lenses: The See-Through Interface. In *Proc. SIGGRAPH*, pages 73–80, New York, 1993. ACM. DOI http://doi.acm.org/10.1145/166117.166126.

[24] K. D. Bilimoria, B. Sridhar, G. B. Chatterji, K. S. Sheth, and S. Grabbe. Facet: Future atm concepts evaluation tool. *Air Traffic Control Quarterly*, 9(1), 2001.

[25] M. J. Black and A. D. Jepson. Eigentracking: Robust matching and tracking of articulated objects using a view-based representation. *International Journal of Computer Vision*, 26(1):63–84, 1998. DOI

10.1023/A:1007939232436. URL http://dx.doi.org/10.1023/A:1007939232436.

[26] R. Boardman. Bubble trees: The visualization of hierarchical information structures. In *Proc. ACM CHI*, pages 315–316, 2000.

[27] J. Böttger, A. Schäfer, G. Lohmann, A. Villringer, and D. Margulies. Three-dimensional mean-shift edge bundling for the visualization of functional connectivity in the brain. *IEEE TVCG*, 20(3):471–480, 2014.

[28] U. Brandes and C. Pich. Eigensolver methods for progressive multidimensional scaling of large data. In *Proc. Graph Drawing*, pages 42–53. Springer LNCS, 2007.

[29] K. Briechle and U. D. Hannebeck. Template matching using fast normalized cross correlation. In *Proc. SPIE*, pages 1–8, 2001.

[30] T. J. Broida and R. Chellappa. Estimation of object motion parameters from noisy images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(1):90–99, Jan 1986. DOI 10.1109/TPAMI.1986.4767755.

[31] S. Bruckner and E. Gröller. Enhancing Depth-Perception with Flexible Volumetric Halos. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1344–1351, November/December 2007. DOI http://doi.ieeecomputersociety.org/10.1109/TVCG.2007.70555.

[32] B. Cabral and L. C. Leedom. Imaging Vector Fields using Line Integral Convolution. In *Proc. SIGGRAPH*, pages 263–270, New York, 1993. ACM. DOI http://doi.acm.org/10.1145/166117.166151.

[33] T. Cao, K. Tang, A. Mohamed, and T. Tan. Parallel banding algorithm to compute exact distance transform with the GPU. In *Proc. ACM SIGGRAPH Symp. on Interactive 3D Graphics and Games*, pages 134–141, 2010. URL http://www.comp.nus.edu.sg/~tants/pba.html.

[34] T. T. Cao, K. Tang, A. Mohamed, and T. S. Tan. Parallel Banding Algorithm to Compute Exact Distance Transform with the GPU. In *Proc. I3D*, pages 134–141, New York, 2010. ACM. DOI http://dx.doi.org/10.1145/1730804.1730818.

[35] M. S. T. Carpendale and C. Montagnese. A Framework for Unifying Presentation Space. In *Proc. UIST*, pages 61–70, New York, 2001. ACM. DOI http://doi.acm.org/10.1145/502348.502358.

[36] D. Castelvecchi. The black box of AI. *Nature*, 538(20), 2016.

[37] L. Cayton. A nearest neighbor data structure for graphics hardware. In *Proc. ADMS*, pages 192–197, 2010. people.kyb.tuebingen.mpg.de/lcayton.

[38] CGAL. CGAL – computational geometry algorithms library, 2013. URL `http://www.cgal.org`.

[39] J. Chai, J. Xiao, and J. Hodgins. Vision-based control of 3D facial animation. In *Proc. EG/SIGGRAPH Symp. on Computer Animation*, 2003.

[40] M. Chalmers. A linear iteration time layout algorithm for visualising high-dimensional data. In *Proc. IEEE Visualization*, pages 127–131, 1996.

[41] W. Y. Chan. A survey on multivariate data visualization, June 2006. Technical Report HKUST/06/2006, Department of Computer Science and Engineering, Hong Kong University of Science and Technology, available at `http://people.stat.sc.edu/hansont/stat730/multivis-report-winnie.pdf`.

[42] D. Chen, A. Farag, R. Falk, and G. Dryden. A variational framework for 3D colonic polyp visualization in virtual colonoscopy. In *Proc. IEEE ICIP*, pages 2617–2620, 2009.

[43] U. Clarenz, M. Rumpf, and A. Telea. Surface processing methods for point sets using finite elements. *Computers and Graphics*, 28(6):851–868, 2004.

[44] D. Coimbra, R. Martins, T. A. T. Neves, A. C. Telea, and F. V. Paulovich. Explaining three-dimensional dimensionality reduction plots. *J. of Information Visualization*, 2015. DOI:10.1177/1473871615600010.

[45] D. B. Coimbra. *Multidimensional Projections for the Visual Exploration of Multimedia Data.* PhD thesis, Johann Bernoulli Institute for Mathematics and Computer Science, University of Groningen, The Netherlands, 2016. URL `http://www.cs.rug.nl/~alext/PAPERS/PhD/coimbra16.pdf`.

[46] D. Comaniciu and P. Meer. Mean shift: A robust approach toward feature space analysis. *IEEE TPAMI*, 24(5):603–619, 2002.

[47] D. Comaniciu, V. Ramesh, and P. Meer. Kernel-based object tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(5): 564–577, May 2003. DOI 10.1109/TPAMI.2003.1195991.

[48] N. D. Cornea, M. F. Demirci, D. Silver, A. Shokoufandeh, and S. Dickinson. 3d object retrieval using many-to-many matching of curve skeletons. In *Proc. Shape Modeling and Applications (SMI'05)*, pages 368–373, New York, 2005. ACM Press.

[49] N. D. Cornea, D. Silver, and P. Min. Curve-skeleton properties, applications, and algorithms. *IEEE TVCG*, 13(3):87–95, 2007.

[50] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *J. Sys. & Software*, 81(12):2252–2268, 2008.

[51] L. Costa and R. Cesar. *Shape Analysis and Classification: Theory and Practice.* CRC Press, 2000.

[52] L. Costa and R. Cesar. *Shape analysis and classification.* CRC Press, 2000.

[53] W. Cui, H. Zhou, H. Qu, P. Wong, and X. Li. Geometry-based edge clustering for graph visualization. *IEEE TVCG*, 14(6):1277–1284, 2008.

[54] D. DeCarlo and A. Santella. Stylization and Abstraction of Photographs. *ACM Transactions on Graphics*, 21(3):769–776, July 2002. DOI http://doi.acm.org/10.1145/566654.566650.

[55] G. DeSouza and A. Kak. Vision for mobile robot navigation: A survey. *IEEE TPAMI*, 24(3):237–250, 2002.

[56] T. Dey and J. Giesen. Detecting undersampling in surface reconstruction. In *Proc.* 7th *annual symposium on Computational geometry (SCG)*, pages 257–263. Springer, 2003.

[57] T. Dey and S. Goswami. Provable surface reconstruction from noisy samples. In *Proc. SCG*, pages 428–438, 2004.

[58] T. Dey, K. Li, E. Ramos, and R. Wenger. Isotopic reconstruction of surfaces with boundaries. *CGF*, 28(5):1371–1382, 2009.

[59] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice Hall, Englewood Cliffs, NJ, 1999.

[60] S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software.* Springer, 2007.

[61] S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software.* Springer, Berlin, 2010.

[62] S. Diehl and A. C. Telea. Multivariate networks in software engineering. In A. K. *et al.*, editor, *Multivariate Network Visualization*, volume 8830, pages 13–36. Springer LNCS, 2013.

[63] C. Distante, G. Diraco, and A. Leone. Active range imaging dataset for indoor surveillance. *Ann. BMVA*, 21(3):1–16, 2010.

[64] A. Dorrington, A. Payne, and M. Cree. An evaluation of time-of-flight range cameras for close range metrology applications. *ISPRS J. Photogramm.*, 38(5):201–206, 2010.

[65] M. van Dortmont, H. van de Wetering, and A. Telea. Skeletonization and distance transforms of 3D volumes using graphics hardware. In *Proc. DGCI*, pages 617–629. Springer LNCS, 2006.

[66] D. J. Duke. Linking Representation with Meaning. In *Posters of IEEE Visualization*, Los Alamitos, 2004. IEEE Computer Society. DOI http://dx.doi.org/10.1109/VISUAL.2004.66.

[67] J. Dykes, A. M. MacEachren, and M. J. Kraak. *Exploring Geovisualization.* Elsevier, 2005.

[68] G. J. Edwards, C. J. Taylor, and T. F. Cootes. Interpreting face images using active appearance models. In *Automatic Face and Gesture Recognition, 1998. Proceedings. Third IEEE International Conference on*, pages 300–305, Apr 1998. DOI 10.1109/AFGR.1998.670965.

[69] M. van Eede, D. Macrini, A. Telea, and C. Sminchisescu. Canonical skeletons for shape matching. In *Proc. ICPR*, pages 542–550, 2006.

[70] G. Elber. Line Illustrations ∈ Computer Graphics. *The Visual Computer*, 11(6):290–296, June 1995. DOI http://dx.doi.org/10.1007/s003710050022.

[71] A. Elgammal, R. Duraiswami, D. Harwood, and L. S. Davis. Background and foreground modeling using nonparametric kernel density estimation for visual surveillance. *Proceedings of the IEEE*, 90(7): 1151–1163, Jul 2002. DOI 10.1109/JPROC.2002.801448.

[72] A. Elgammal, D. Harwood, and L. Davis. Non-parametric model for background subtraction. In D. Vernon, editor, *Computer Vision — ECCV 2000: 6th European Conference on Computer Vision Dublin, Ireland, June 26–July 1, 2000 Proceedings, Part II*, pages 751–767, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-45053-5. DOI 10.1007/3-540-45053-X_48. URL `http://dx.doi.org/10.1007/3-540-45053-X_48`.

[73] G. Ellis and A. Dix. A taxonomy of clutter reduction for information visualisation. *IEEE TVCG*, 13(6):1216–1223, 2007.

[74] N. Elmqvist, P. Dragicevic, and J. Fekete. Rolling the dice: Multidimensional visual exploration using scatterplot matrix navigation. *IEEE TVCG*, 14(6):1539–1148, 2008.

[75] O. Ersoy, C. Hurter, F. Paulovich, G. Cantareira, and A. Telea. Skeleton-Based Edge Bundling for Graph Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2364–2373, December 2011. DOI http://dx.doi.org/10.1109/TVCG.2011.233.

[76] O. Ersoy, C. Hurter, F. Paulovich, G. Cantareiro, and A. Telea. Skeleton-based edge bundles for graph visualization. *IEEE TVCG*, 17 (2):2364–2373, 2011.

[77] Eurocontrol. NEST: Network Strategic Tool, 2013. `www.eurocontrol.int/articles/airspace-modelling`.

[78] M. H. Everts, H. Bekker, J. Roerdink, and T. Isenberg. Depth-dependent halos: Illustrative rendering of dense line data. *IEEE TVCG*, 15(6):1299–1306, 2009.

[79] M. H. Everts, H. Bekker, J. B. T. M. Roerdink, and T. Isenberg. Depth-Dependent Halos: Illustrative Rendering of Dense Line Data. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1299–1306, November/December 2009. DOI http://doi.ieeecomputersociety.org/10.1109/TVCG.2009.138.

[80] S. G. Fadel, F. M. Fatore, F. S. Duarte, and F. V. Paulovich. LoCH: a neighborhood-based multidimensional projection technique for high-dimensional sparse spaces. *Neurocomputing*, 150, Part B(0):546 – 556, 2015.

[81] A. X. Falcão, J. K. Udupa, and F. K. Miyazawa. An ultra-fast user-steered image segmentation paradigm: live wire on the fly. *IEEE Trans Med Imaging*, 19(1):55–62, 2000.

[82] A. X. Falcão, J. Stolfi, and R. A. Lotufo. The image foresting transform: Theory, algorithms, and applications. *IEEE TPAMI*, 26(1):19–29, 2004.

[83] C. Faloutsos and K. Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. *ACM SIGMOD Record*, 2(2):163–174, 1995.

[84] C. Feng, A. Jalba, and A. Telea. A descriptor for voxel shapes based on the skeleton cut space. In *Proc. Eurographics Workshop on 3D Object Retrieval (3DOR)*. Eurographics, 2016.

[85] P. Fieguth and D. Terzopoulos. Color-based tracking of heads and other mobile objects at video frame rates. In *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, pages 21–27, Jun 1997. DOI 10.1109/CVPR.1997.609292.

[86] Finviz. Map of the market, 2016. URL `https://finviz.com/map.ashx`.

[87] M. Foskey, M. Lin, and D. Manocha. Efficient computation of a simplified medial axis. In *Proc. Shape Modeling*, pages 135–142, 2003.

[88] A. Frank and A. Asuncion. UCI machine learning repository, 2013. URL `http://www.ics.uci.edu/~mlearn`.

[89] J. Freixenet, X. Munoz, D. Raba, J. Marti, and X. Cufi. Yet another survey on image segmentation: Region and boundary information integration. In *Proc. ECCV*, pages 408–422, 2002.

[90] K. S. Fu and J. K. Mui. A survey on image segmentation. *Pattern Recognition*, 13(1):3–16, 1981.

[91] GAIN Group. Guide to methods and tools for airline flight safety analysis, 2004. `flightsafety.org/files/analytical_methods_and_tools.pdf`, 2^{nd} ed.

[92] E. Gansner, Y. Hu, S. North, and C. Scheidegger. Multilevel agglomerative edge bundling for visualizing large graphs. In *Proc. PacificVis*, pages 187–194, 2011.

[93] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using GPU. In *Proc. Intl. Workshop on Computer Vision on GPU (CVGPU)*, pages 77–83, 2008.

[94] H. Gaspard-Boulinc, Y. Jestin, and L. Fleury. Epoques: a user-centerd approach to design tools and methods for atm safety occurences treatment. In *Proc. ESReDA (European Safety, Reliability and Data Association)*. European Commission, 2003.

[95] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Mach Learn*, 63(1):3–42, 2006. ISSN 0885-6125.

[96] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik. Gene selection for cancer classification using support vector machines. *Machine Learning*, 46(1-3):389–422, 2002. ISSN 0885-6125.

[97] Z. H, X. Yuan, H. Qu, W. Cui, and B. Chen. Visual clustering in parallel coordinates. *CGF*, pages 1324–1332, 2008.

[98] F. van Ham and M. Wattenberg. Centrality based visualization of small world graphs. *CGF*, 27(3):972–980, 2008.

[99] F. van Ham and J. J. van Wijk. Interactive visualization of small world graphs. In *Proc. IEEE Infovis*, pages 199–206, 2004.

[100] D. Hand, H. Mannila, and P. S. P. *Principles of Data Mining*. MIT Press, 2001.

[101] C. Hansen and C. J. Johnson. *The Visualization Handbook*. Elsevier, Amsterdam, 2005.

[102] C. Harris and M. Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, page 50. Citeseer, 1988.

[103] M. Hassouna and A. Farag. Variational curve skeletons using gradient vector flow. *IEEE TPAMI*, 31(12):2257–2274, 2009.

[104] S. Havre, E. Hetzler, P. Whitney, and L. Nowell. ThemeRiver: visualizing thematic changes in large document collections. *IEEE TVCG*, 8 (1):9–20, 2002.

[105] A. Helgeland and O. Andreassen. Visualization of Vector Fields using Seed LIC and Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 10(6):673–682, November/December 2004. DOI http://dx.doi.org/10.1109/TVCG.2004.49.

[106] W. Hesselink and J. Roerdink. Euclidean skeletons of digiral image and volume data in linear time by the integer medial axis transform. *IEEE TPAMI*, 30(12):2204–2217, 2008.

[107] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE TVCG*, 12(5):741–748, 2006.

[108] D. Holten and J. J. van Wijk. Force-directed edge bundling for graph visualization. *Computer Graphics Forum*, 28(3):670–677, 2009.

[109] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganized points. *Proc. ACM SIGGRAPH*, 26(2):71–78, 1992.

[110] M. Hovinen, A. Aisla, and S. Pyörälä. Visual detection of technical success and effectiveness of teat cleaning in two automatic milking systems. *J. Dairy Sci.*, (88):3354–3362, 2005.

[111] A. Hunt. Teat detection for an automatic milking system. In *MSc thesis, Univ. of Dublin, Ireland*, 2006. `doras.dcu.ie/17194/1/aidan_hunt_duffy_20120703135817.pdf`.

[112] C. Hurter, B. Tissoires, and S. Conversy. FromDaDy: Spreading data across views to support iterative exploration of aircraft trajectories. *IEEE TVCG*, 15(6):1017–1024, 2009.

[113] C. Hurter, B. Tissoires, and S. Conversy. FromDaDy: Spreading data across views to support iterative exploration of aircraft trajectories. *IEEE TVCG*, 15(6):1017–1024, 2009.

[114] C. Hurter, O. Ersoy, and A. Telea. MoleView: An attribute and structure-based semantic lens for large element-based plots. *IEEE TVCG*, 17(12):2600–2609, 2011.

[115] C. Hurter, O. Ersoy, and A. Telea. MoleView: An attribute and structure-based semantic lens for large element-based plots. *IEEE TVCG*, 17(12):2600–2609, 2011.

[116] C. Hurter, O. Ersoy, and A. Telea. Graph bundling by kernel density estimation. *Computer Graphics Forum*, 31(3):435–443, 2012.

[117] C. Hurter, O. Ersoy, and A. Telea. Smooth bundling of large streaming and sequence graphs. In *Proc. PacificVis*, pages 41–48. IEEE Press, 2013.

[118] C. Hurter, S. Conversy, D. Gianazza, and A. Telea. Interactive image-based information visualization for aircraft trajectory analysis. *Transportation Research Part C: Emerging Technologies*, 48, 2014.

[119] C. Hurter, O. Ersoy, S. Fabrikant, T. Klein, and A. Telea. Bundled visualization of dynamic graph and trail data. *IEEE TVCG*, 20(8):1141–1157, 2014.

[120] C. Hurter, R. Taylor, S. Carpendale, and A. Telea. Color tunneling: Interactive exploration and selection in volumetric datasets. In *Proc. IEEE PacificVis*, pages 322–330, 2014.

[121] C. Hurter, S. Puechmorel, F. Nicol, and A. Telea. Functional decomposition for bundled simplification of trail sets. *IEEE TVCG*, 24(1): 500–510, 2018.

[122] C. Hurter. *Image-Based Visualization: Interactive Multidimensional Data Exploration.* Synthesis Lectures on Visualization. Morgan & Claypool, 2015.

[123] C. Hurter, A. Telea, and O. Ersoy. MoleView: An Attribute and Structure-Based Semantic Lens for Large Element-Based Plots. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2600–2609, December 2011. DOI http://dx.doi.org/10.1109/TVCG.2011.223.

[124] S. Ingram, T. Munzner, and M. Olano. Glimmer: Multilevel MDS on the GPU. *Visualization and Computer Graphics, IEEE Transactions on*, 15(2):249–261, March 2009.

[125] A. Inselberg and B. Dimsdale. Parallel coordinates: A tool for visualizing multidimensional geometry. In *Proc. IEEE Visualization*, pages 361–378, Los Alamitos, CA, 1990. IEEE Press.

[126] V. Interrante and C. Grosch. Visualizing 3D Flow. *IEEE Computer Graphics & Applications*, 18(4):49–53, July 1998. DOI http://dx.doi.org/10.1109/38.689664.

[127] A. S. Jackson, A. Bulat, V. Argyriou, and G. Tzimiropoulos. Large pose 3D face reconstruction from a single image via direct volumetric CNN regression. In *Proc. ICCV*, 2017.

[128] E. D. Jansing, T. A. Albert, and D. L. Chenoweth. Two-dimensional entropic segmentation. *Pattern Recognition Letters*, 20(3):329–336, 1999.

[129] B. Jobard and W. Lefer. Creating evenly-spaced streamlines of arbitrary density. In *Proc. EG Workshop on Visualization in Scientific Computing*, pages 43–56, New York, 1997. Springer.

[130] P. Joia, D. Coimbra, J. A. Cuminato, F. V. Paulovich, and L. G. Nonato. Local affine multidimensional projection. *IEEE TVCG*, 17(12): 2563–2571, 2011.

[131] I. T. Jolliffe. *Principal Component Analysis.* Springer, 2002.

[132] F. Jourdan and G. Melançon. Multiscale hybrid MDS. In *Information Visualisation*, pages 388–393, 2004.

[133] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1(4):321–331, 1988.

[134] M. Kazhdan. Reconstruction of solid models from oriented point sets. In *Proc. EG Symp. on Geometry Processing (SGP)*, pages 163–171, 2005.

[135] M. Kazhdan, M. Bolitho, and H. Hoppe. Poisson surface reconstruction. In *Proc. Symp. Geometry Processing (SGP)*, pages 61–70, 2006. `http://www.cs.jhu.edu/~misha/Code/PoissonRecon/Version4.51`.

[136] D. Keim, G. Andrienko, N. Andrienko, P. Jankowski, M. Kraak, A. MacEachren, and S. Wrobel. Geovisual analytics for spatial decision support: Setting the research agenda. *Intl. J. Geo. Info. Sci.*, 21 (8):839–857, 2007.

[137] T. Klein, M. van der Zwan, and A. Telea. Dynamic multiscale visualization of flight data. In S. Battiato and J. Braz, editors, *Proc. of the 9th IEEE International Conference on Computer Vision Theory and Applications (VISAPP)*, volume 1, pages 232–240, 2014.

[138] C. Koning and J. Rodenburg. Automatic milking: State of the art in Europe and North America. *International Information Systems for Agricultural Science and Technology*, 22(12), 2004.

[139] R. Krüger, D. Thom, M. Wörner, H. Bosch, and T. Ertl. TrajectoryLenses - a set-based filtering and exploration technique for long-term trajectory data. *CGF*, 32(3):451–460, 2013.

[140] J. B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29:115–129, 1964.

[141] J. Kustra, A. Jalba, and A. Telea. Robust segmentation of multiple intersecting manifolds from unoriented noisy point clouds. *CGF*, 33 (1):73–87, 2014.

[142] A. Lambert, R. Bourqui, and D. Auber. Winding roads: Routing edges into bundles. *CGF*, 29(3):432–439, 2010.

[143] A. Lambert, R. Bourqui, and D. Auber. 3D edge bundling for geographical data visualization. In *Proc. Information Visualisation*, pages 329–335, 2010.

[144] O. Lampe and H. Hauser. Interactive visualization of streaming data with kernel density estimation. In *Proc. IEEE PacificVis*, pages 232–239, 2011.

[145] N. Lazaros, G. Sirakoulis, and A. Gasteratos. Review of stereo vision algorithms: From software to hardware. *Int. J. Optomechatronics*, 2: 435–462, 2008.

[146] D. J. Lehmann, G. Albuquerque, M. Eisemann, M. Magnor, and H. Theisel. Selecting coherent and relevant plots in large scatterplot matrices. *Computer Graphics Forum*, 31(6):1895–1908, 2012.

[147] Lely Technologies BV. Automatic milking robotic devices for the dairy industry, 2017. `www.lely.com`.

[148] S. Lespinats and M. Aupetit. CheckViz: Sanity check and topological clues for linear and non-linear mappings. *Computer Graphics Forum*, 30(1):113–125, 2011.

[149] C. Letondal, C. Hurter, R. Lesbordes, J. L. Vinot, and S. Conversy. Flights in my hands: coherence concerns in designing strip'tic, a tangible space for air traffic controllers. In *Proc. ACM CHI*, pages 2175–2184, 2013.

[150] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[151] J. Lewis. Fast normalized cross-correlation. *Vision interface*, 10(1): 120–123, 1995.

[152] A. Lhuillier, C. Hurter, and A. Telea. State of the art in edge and trail bundling techniques. *Computer Graphics Forum*, 36(3):619–645, 2017.

[153] A. Lhuillier, C. Hurter, and A. Telea. FFTEB: edge bundling of huge graphs by the Fast Fourier Transform. In *Proc. IEEE PacificVis*, 2017.

[154] C. Li, C. Xu, C. Gui, and M. D. Fox. Distance regularized level set evolution and its application to image segmentation. *IEEE TPAMI*, 19 (12):3243–3254, 2010.

[155] L. Li and H. W. Shen. Image-based streamline generation and rendering. *IEEE TVCG*, 13(3):630–640, 2007.

[156] L. Li and M. K. H. Leung. Integrating intensity and texture differences for robust change detection. *IEEE Transactions on Image Processing*, 11(2):105–112, Feb 2002. DOI 10.1109/83.982818.

[157] H. Lipson and M. Kurman. *Fabricated: The new world of 3D printing*. Addison-Wesley, 2013.

[158] O. Litany, A. Bronstein, M. Bronstein, and A. Makadia. Deformable shape completion with graph convolutional autoencoders, 2017. arXiv:1712.00268.

[159] LMI Technologies. Time of flight imaging enables automated milking, 2012. `www.lmi3d.com`.

[160] D. Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 60(2):91–110, 2004.

[161] W. Lueks, I. Viola, M. van der Zwan, H. Bekker, and T. Isenberg. Spatially Continuous Change of Abstraction in Molecular Visualization. In *Abstracts of IEEE BioVis*, Los Alamitos, 2011. IEEE Computer Society. URL `http://www.biovis.net/2011/papers_abstracts/abstracts/131.html`.

[162] L. van der Maaten and G. Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2431–2456, 2008.

[163] L. van der Maaten and G. Hinton. Visualizing non-metric similarities in multiple maps. *Machine Learning*, 87(1):33–35, 2012.

[164] L. van der Maaten, E. Postma, and H. van den Herik. Dimensionality reduction: A comparative review. *Journal of Machine Learning Research*, 10(1):66–71, 2009.

[165] D. Macrini, K. Siddiqi, and S. Dickinson. From skeletons to bone graphs: Medial abstraction for object recognition. In *Proc. CVPR*, pages 324–332, 2008.

[166] M. Maire, P. Arbeláez, C. Fowlkes, and J. Malik. Using contours to detect and localize junctions in natural images. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[167] F. Mansmann, D. A. Keim, S. C. North, B. Rexroad, and D. Sheleheda. Visual analysis of network traffic for resource planning, interactive monitoring, and interpretation of security threats. *IEEE TVCG*, 13(6): 985–997, 2007.

[168] R. Martins, D. Coimbra, R. Minghim, and A. Telea. Visual analysis of dimensionality reduction quality for parameterized projections. *Computers & Graphics*, 41:26–42, 2014.

[169] R. Martins, R. Minghim, and A. Telea. Explaining neighborhood preservation for multidimensional projections. In *Proc. Computer Graphics and Visual Computing (CGVC)*, pages 121–128. Eurographics, 2015.

[170] R. M. Martins. *Explanatory Visualization of Multidimensional Projections*. PhD thesis, Johann Bernoulli Institute for Mathematics and Computer Science, University of Groningen, The Netherlands, 2016. URL `http://www.cs.rug.nl/~alext/PAPERS/PhD/martins16.pdf`.

[171] A. Marzuoli, C. Hurter, and E. Feron. Data visualization techniques for airspace flow modeling. In *Proc. Intelligent Data Understanding (CIDU)*, pages 79–86, 2012.

[172] B. H. McCormick, T. A. DeFanti, and M. D. Brown. Visualization in scientific computing. *Computer Graphics*, 21(6), 1987.

[173] F. McGee and J. Dingliana. An empirical study on the impact of edge bundling on user comprehension of graphs. In *Proc. AVI*, pages 620–627, 2012.

[174] A. Mebarki, P. Alliez, and O. Devillers. Farthest point seeding for efficient placement of streamlines. In *Proc. IEEE Visualization*, pages 479–486, Los Alamitos, CA, 2005. IEEE CS Press.

[175] N. L. of Medicine. The insight segmentation and registration toolkit (itk), 2014. URL `http://www.itk.org`.

[176] N. Meinshausen and P. Bühlmann. Stability selection. *J. Royal Stat Soc: Series B*, 72(4):417–473, 2010. ISSN 1467-9868.

[177] Mesa Imaging. SR4000 user manual, 2010. `www.mesa-imaging.ch/prodview4k.php`.

[178] MESA Imaging. Automatic milking systems, 2014. `www.mesa-imaging.ch/applications/automatic-milking-systems`.

[179] X. Mi, D. DeCarlo, and M. Stone. Abstraction of 2D Shapes in Terms of Parts. In *Proc. NPAR*, pages 15–24, New York, 2009. ACM. DOI http://doi.acm.org/10.1145/1572614.1572617.

[180] Microsoft. Age of empires trilogy, 2003. URL `http://www.microsoft.com/games/empires`.

[181] T. B. Moeslund, G. Thomas, and A. Hilton. *Computer Vision in Sports*. Springer, 2014.

[182] B. Moghaddam and A. Pentland. Probabilistic visual learning for object representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(7):696–710, Jul 1997. DOI 10.1109/34.598227.

[183] H. P. Moravec. Visual mapping by a robot rover. In *Proceedings of the 6th international joint conference on Artificial intelligence-Volume 1*, pages 598–600. Morgan Kaufmann Publishers Inc., 1979.

[184] D. Moura. 3D density histograms for criteria-driven edge bundling, 2015. *arXiv:1504.02687v1* [cs.GR].

[185] T. Munzner. *Visualization Analysis and Design*. CRC Press, 2015.

[186] P. Neumann, T. Isenberg, and S. Carpendale. NPR Lenses: Interactive Tools for Non-Photorealistic Line Drawings. In *Proc. Smart Graphics*, pages 10–22, Berlin, Heidelberg, 2007. Springer-Verlag. DOI http://dx.doi.org/10.1007/978-3-540-73214-3_2.

[187] Q. Nguyen, P. Eades, and S. H. Hong. StreamEB: Stream edge bundling. In *Proc. Graph Drawing*, pages 324–332. Springer, 2012.

[188] M. Nollenburg and A. Wolff. Drawing and labeling high-quality metro maps by mixed-integer programming. *IEEE TVCG*, 17(5):626–641, 2010.

[189] R. L. Ogniewicz and O. Kubler. Hierarchic voronoi skeletons. *Patt. Recog.*, (28):343– 359, 1995.

[190] OpenCV. The opencv computer vision library, 2014. URL `http://opencvlibrary.sourceforge.net/`.

[191] N. Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man and Cybernetics*, 9(1):62–66, 1979.

[192] A. C. Oztireli, G. Guennebaud, and M. Gross. Feature preserving point set surfaces based on nonlinear kernel regression. *Computer Graphics Forum*, 28(2):493–501, 2009.

[193] J. Paiva, W. Schwartz, H. Pedrini, and R. Minghim. Semi-supervised dimensionality reduction based on partial least squares for visual analysis of high dimensional data. *Computer Graphics Forum*, 31(3): 1345–1354, 2012.

[194] K. Palagyi and A. Kuba. Directional 3D thinning using 8 subiterations. In *Proc. DGCI*, volume 1568, pages 325–336. Springer LNCS, 1999.

[195] G. Papari, N. Petkov, and P. Campisi. Artistic edge and corner preserving smoothing. *IEEE TIP*, 16(10):2449–2462, 2007.

[196] N. Paragios and R. Deriche. Geodesic active regions and level set methods for supervised texture segmentation. *International Journal of Computer Vision*, 46(3):223–247, 2002. DOI 10.1023/A:1014080923068. URL `http://dx.doi.org/10.1023/A:1014080923068`.

[197] S. Park and J. K. Aggarwal. A hierarchical bayesian network for event recognition of human actions and interactions. *Multimedia Syst.*, 10 (2):164–179, August 2004. DOI 10.1007/s00530-004-0148-1. URL `http://dx.doi.org/10.1007/s00530-004-0148-1`.

[198] S. W. Park, L. Linsen, O. Kreylos, J. D. Owens, and B. Hamann. Discrete Sibson interpolation. *IEEE TVCG*, 12(2):243–253, 2006.

[199] F. Paulovich, D. Eler, J. Poco, C. Botha, R. Minghim, and L. G. Nonato. Piecewise Laplacian-based projection for interactive data exploration and organization. *Computer Graphics Forum*, 30(3):1091–1100, 2011.

[200] F. V. Paulovich and R. Minghim. HiPP: A novel hierarchical point placement strategy and its application to the exploration of document collections. *IEEE TVCG*, 14(6):1229–1236, 2008.

[201] F. V. Paulovich, L. G. Nonato, and R. Minghim. Visual mapping of text collections through a fast high precision projection technique. In *Information Visualization (IV), Tenth International Conference on*, pages 282–290, 2006.

[202] F. V. Paulovich, L. G. Nonato, R. Minghim, and H. Levkowitz. Least square projection: A fast high precision multidimensional projection technique and its application to document mapping. *IEEE TVCG*, 14 (3):564–575, 2008.

[203] F. V. Paulovich, C. Silva, and L. G. Nonato. Two-phase mapping for projecting massive data sets. *IEEE TVCG*, 16:1281–1290, 2010.

[204] F. V. Paulovich, F. M. B. Toledo, G. P. Telles, R. Minghim, and L. G. Nonato. Semantic wordification of document collections. *Computer Graphics Forum*, 31(3pt3):1145–1153, 2012.

[205] J. Pei, J. Han, R. Mao, et al. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD workshop on research issues in data mining and knowledge discovery*, volume 4, 2000.

[206] V. Peysakhovich, C. Hurter, and A. Telea. Attribute-driven edge bundling for general graphs with applications in trail analysis. In *Proc. IEEE PacificVis*, 2015.

[207] PlaneFinder. Live flight status tracker, 2016. `http://planefinder.net`.

[208] C. Pudney. Distance-ordered homotopic thinning: A skeletonization algorithm for 3D digital images. *CVIU*, 72(3):404–413, 1998.

[209] S. Pupyrev, L. Nachmanson, S.Bereg, and E. Holroyd. Edge routing with ordered bundles. In *Proc. Graph Drawing*, pages 136–147, 2012.

[210] R. Rao and S. K. Card. The table lens: Merging graphical and symbolic representations in an interactive focus+context visualization for tabular information. In *Proc. ACM Conference on Human Factors in Computing Systems (CHI)*, pages 318–322, New York, 1994. ACM Press.

[211] P. E. Rauber, A. X. Falcão, T. V. Spina, and P. J. de Rezende. Interactive segmentation by image foresting transform on superpixel graphs. In *Proc. SIBGRAPI*, pages 133–140, Los Alamitos, CA:, 2013. IEEE Press.

[212] P. E. Rauber, S. G. Fadel, A. X. Falcão, and A. C. Telea. Visualizing the hidden activity of artificial neural networks. *IEEE TVCG*, 2016. DOI 10.1109/TVCG.2016.2598838.

[213] P. E. Rauber, R. R. O. Silva, S. Feringa, M. E. Celebi, A. X. Falcão, and A. C. Telea. Interactive Image Feature Selection Aided by Dimensionality Reduction. In *Proc. EuroVis Workshop on Visual Analytics (EuroVA)*. The Eurographics Association, 2015.

[214] P. Rauber, A. Falcão, and A. Telea. Visualizing time-dependent data using dynamic t-SNE. In *Proc. EuroVis – Short Papers*, 2016.

[215] P. Rautek, S. Bruckner, E. Gröller, and I. Viola. Illustrative Visualization: New Technology or Useless Tautology? *ACM SIGGRAPH Computer Graphics*, 42(3):4:1–4:8, August 2008. DOI http://doi.acm.org/10.1145/1408626.1408633.

[216] D. Reniers and A. Telea. Tolerance-based feature transforms. In *Advances in Comp. Graphics and Comp. Vision (eds. J. Jorge* et al.*)*, pages 187–200. Springer, 2007.

[217] D. Reniers, J. J. van Wijk, and A. Telea. Computing Multiscale Skeletons of Genus 0 Objects using a Global Importance Measure. *IEEE TVCG*, 14(2):355–368, March/April 2008. DOI http://dx.doi.org/10.1109/TVCG.2008.23.

[218] D. Reniers, L. Voinea, O. Ersoy, and A. Telea. The Solid* toolset for software visual analytics of program structure and metrics comprehension: From research prototype to product. *Science of Computer Programming*, 79(1):224–240, 2014.

[219] J. Rittscher, J. Kato, S. Joga, and A. Blake. A probabilistic background model for tracking. In D. Vernon, editor, *Computer Vision — ECCV 2000: 6th European Conference on Computer Vision Dublin, Ireland, June 26–July 1, 2000 Proceedings, Part II*, pages 336–350, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-45053-5. DOI 10.1007/3-540-45053-X_22. URL `http://dx.doi.org/10.1007/3-540-45053-X_22`.

[220] J. B. T. M. Roerdink and A. Meijster. The watershed transform: Definitions, algorithms and parallelization strategies. *Fundamenta Informaticae*, 41:187–228, 2001.

[221] R. Rosales and S. Sclaroff. 3d trajectory recovery for tracking multiple objects and trajectory guided recognition of actions. In *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on.*, volume 2, pages 117–123. IEEE, 1999.

[222] S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, 2000.

[223] M. Rumpf and A. Telea. A continuous skeletonization method based on level sets. In *Proc. VisSym*, pages 151–158, 2002.

[224] V. Salari and I. K. Sethi. Feature point correspondence in the presence of occlusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(1):87–91, Jan 1990. DOI 10.1109/34.41387.

[225] R. Scheepens, N. Willems, H. van de Wetering, G. Andrienko, N. Andrienko, and J. J. van Wijk. Composite density maps for multivariate trajectories. *IEEE TVCG*, 17(12):2518–2527, 2011.

[226] C. Schmid, R. Mohr, and C. Bauckhage. Evaluation of interest point detectors. *International Journal of Computer Vision*, 37(2):151–172, 2000. DOI 10.1023/A:1008199403446.

[227] W. Schroeder, J. Zarge, and W. Lorensen. Decimation of triangle meshes. *Proc. SIGGRAPH '92, Computer Graphics*, 26(2):65–70, 1992.

[228] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3-D Graphics, 4^{rd} edition*. Kitware, Inc., Clifton Park, NY:, 2006.

[229] Scott Milktech Ltd. World's first automatic milking system, 2013. `scott.co.nz/scott-milktech`.

[230] M. Sedlmair, T. Munzer, and M. Tory. Empirical guidance on scatterplot and dimension reduction technique choices. *IEEE TVCG*, 19(12): 2634–2643, 2013.

[231] D. Selassie, B. Heller, and J. Heer. Divided edge bundling for directional network data. *IEEE TVCG*, 19(12):754–763, 2011.

[232] D. Serby, E. K. Meier, and L. V. Gool. Probabilistic object tracking using multiple features. In *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, volume 2, pages 184–187, 2004.

[233] I. K. Sethi and R. Jain. Finding trajectories of feature points in a monocular image sequence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9(1):56–73, Jan 1987. DOI 10.1109/TPAMI.1987.4767872.

[234] J. A. Sethian. *Level Set Methods: Evolving Interfaces in Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University Press, Cambridge, UK, 1996.

[235] D. Shaked and A. Bruckstein. Pruning medial axes. *CVIU*, 69(2):156–169, 1998.

[236] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE TPAMI*, 22(8):888–905, 2000.

[237] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE TPAMI*, 22(8):888–905, 2000.

[238] J. Shi and C. Tomasi. Good features to track. In *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 593–600, Jun 1994. DOI 10.1109/CVPR.1994.323794.

[239] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proc. ACM VL*, pages 336–343, 1996.

[240] K. Siddiqi and S. Pizer. *Medial Representations: Mathematics, Algorithms and Applications.* Springer, 2009.

[241] K. Siddiqi, S. Bouix, A. Tannenbaum, and S. Zucker. Hamilton-Jacobi skeletons. *IJCV*, 48(3):215–231, 2002.

[242] C. Silva, F. Paulovich, and L. G. Nonato. User-centered multidimensional projection techniques. *Comput. Sci. Eng.*, 14(4):74–81, 2012.

[243] R. R. O. da Silva, P. Rauber, R. M. Martins, R. Minghim, and A. Telea. Attribute-based visual explanation of multidimensional projections. In *Proc. EuroVis Workshop on Visual Analytics (EuroVA)*, pages 137–142, 2015.

[244] R. R. O. d. Silva, P. E. Rauber, R. M. Martins, R. Minghim, and A. C. Telea. Attribute-based Visual Explanation of Multidimensional Projections. In E. Bertini and J. C. Roberts, editors, *EuroVis Workshop on Visual Analytics (EuroVA).* The Eurographics Association, 2015.

[245] R. R. O. da Silva, E. F. Vernier, P. E. Rauber, J. L. D. Comba, R. Minghim, and A. Telea. Metric Evolution Maps: Multidimensional attribute-driven exploration of software repositories. In *Proc.* 21st *International Symposium on Vision, Modeling and Visualization (VMV).* Eurographics, 2016.

[246] V. de Silva and J. Tenenbaum. Sparse multidimensional scaling using landmark points. Technical report, Stanford Univ., 2004. `window.stanford.edu/courses/cs468-05-winter/Papers/Landmarks/Silva_landmarks5.pdf`. Last access: 15/07/2013.

[247] V. D. Silva and J. B. Tenenbaum. Global versus local methods in nonlinear dimensionality reduction. In *Advances in Neural Information Processing Systems*, volume 15, pages 705–712. MIT Press, 2002.

[248] C. Sminchisescu and B. Triggs. Covariance scaled sampling for monocular 3D body tracking. In *Proc. IEEE CVPR*, pages 447–454, 2001.

[249] C. Sorzano, J. Vargas, and A. Pascual-Montano. A survey of dimensionality reduction techniques, 2014. *arxiv.org/pdf/1403.2877.*

[250] D. Stalling and H. C. Hege. Fast and Resolution Independent Line Integral Convolution. In *Proc. SIGGRAPH*, pages 249–256, New York, 1995. ACM. DOI http://dx.doi.org/10.1145/218380.218448.

[251] C. Stauffer and W. E. L. Grimson. Learning patterns of activity using real-time tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):747–757, Aug 2000. DOI 10.1109/34.868677.

[252] B. Stenger, V. Ramesh, N. Paragios, F. Coetzee, and J. M. Buhmann. Topology free hidden markov models: application to background modeling. In *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, volume 1, pages 294–301 vol.1, 2001. DOI 10.1109/ICCV.2001.937532.

[253] S. Stolpner, S. Whitesides, and K. Siddiqi. Sampled medial loci and boundary differential geometry. In *Proc. IEEE 3DIM*, pages 87–95, 2009.

[254] J. Stott, P. Rodgers, J. Martinez-Ovando, and S. Walker. Automatic metro map layout using multicriteria optimization. *IEEE TVCG*, 17 (1):101–114, 2011.

[255] R. Strzodka and A. Telea. Generalized distance transforms and skeletons in graphics hardware. In *Proc. VisSym*, pages 221–230, 2004.

[256] A. Sud, M. Foskey, and D. Manocha. Homotopy-preserving medial axis simplification. In *Proc. SPM*, pages 103–110, 2005.

[257] H. Sundar, D. Silver, N. Gagvani, and S. Dickinson. Skeleton based shape matching and retrieval. In *Proc. SMI*, pages 130–138, 2003.

[258] M. Sussman and G. Wright. The correlation coefficient technique for pattern matching. In *Proc. ISMRM*, page 203, 1999.

[259] A. Tagliasacchi, T. Delame, M. Spagnuolo, N. Amenta, and A. Telea. 3D skeletons: A state-of-the-art report. *CGF*, 35(2):573–597, 2016.

[260] P. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.

[261] J. W. Tangelder and R. Veltkamp. A survey of content based 3D shape retrieval methods. *Multimed Tools Appl*, 39(3):441–471, 2008.

[262] M. Tarini, P. Cignoni, and C. Montani. Ambient Occlusion and Edge Cueing to Enhance Real Time Molecular Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1237–884, September/October 2006. DOI http://doi.ieeecomputersociety.org/10.1109/TVCG.2006.115.

[263] E. Tejada, R. Minghim, and L. G. Nonato. On improved projection techniques to support visual exploration of multidimensional data sets. *J. of Information Visualization*, 2(4):218–231, 2003.

[264] A. Telea. Combining extended table lens and treemap techniques for visualizing tabular data. In *Proc. EuroVis*, pages 51–58, 2006.

[265] A. Telea. CUDA Skeletonization and Distance Transform Toolkit, 2011. URL `http://www.cs.rug.nl/~alext/CUDASKEL`. http://www.cs.rug.nl/~alext/CUDASKEL.

[266] A. Telea. Feature preserving smoothing of shapes using saliency skeletons. *Visualization in Medicine and Life Sciences*, pages 155–172, 2012.

[267] A. Telea and O. Ersoy. Image-based edge bundles: Simplified visualization of large graphs. *Computer Graphics Forum*, 29(3):543–551, 2010.

[268] A. Telea and J. J. van Wijk. An augmented fast marching method for computing skeletons and centerlines. In *Proc. VisSym*, pages 251–259, 2002.

[269] A. Telea and J. J. van Wijk. 3D IBFV: Hardware-accelerated 3d flow visualization. In *Proc. IEEE Visualization*, pages 31–38, Los Alamitos, CA, 2003. IEEE Press.

[270] A. Telea, H. Hoogendorp, O. Ersoy, and D. Reniers. Extraction and visualization of call dependencies for large C/C++ code bases: A comparative study. In *Proc. IEEE VISSOFT*, pages 81–88, 2009.

[271] A. C. Telea. *Data visualization – Principles and Practice, 2^{nd} edition.* CRC Press, 2014.

[272] J. B. Tenenbaum, V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290 (5500):2319–2323, 2000.

[273] M. L. Terpstra. Dense skeletons for image compression and manipulation. In *MSc thesis, Institute Johann Bernoulli, Univ. of Groningen, Netherlands*, 2017. `scripties.fwn.eldoc.ub.rug.nl/scripties/Informatica/Master/2017/Terpstra.M.L.`

[274] Thales, Inc. CoFlight Flight Data Processing Framework, 2013. `www.thalesgroup.com`.

[275] H. Theisel, T. Weinkauf, H. C. Hege, and H. P. Seidel. Saddle Connectors – An Approach to Visualizing the Topological Skeleton of Complex 3D Vector Fields. In *Proc. IEEE Visualization*, pages 225–232, Los Alamitos, 2003. IEEE Computer Society. DOI http://dx.doi.org/10.1109/VISUAL.2003.1250376.

[276] J. A. Thomas and K. A. Cook, editors. *Illuminating the Path: Research and Development Agenda for Visual Analytics.* IEEE Press, Los Alamitos, CA, 2005.

[277] O. J. Tobias and R. Seara. Image segmentation by histogram thresholding using fuzzy sets. *IEEE Trans. on Image Processing*, 11(12):1457–1465, 2002.

[278] W. S. Torgeson. Multidimensional scaling of similarity. *Psychometrika*, 30:379–393, 1965.

[279] E. R. Tufte. *The Visual Display of Quantitative Information, 2nd edition.* Graphics Press, Cheshire, CT, 2001.

[280] J. W. Tukey and P. A. Tukey. Computer graphics and exploratory data analysis: An introduction. *The Collected Works of John W. Tukey: Graphics: 1965-1985*, 5:419, 1988.

[281] G. Turk and D. Banks. Image-guided streamline placement. In *Proceedings of SIGGRAPH 96, Computer Graphics Proceedings, Annual Conference Series*, pages 453–460, Reading, MA, 1996. Addison Wesley.

[282] R. van Liere and W. de Leeuw. Graphsplatting: Visualizing graphs as continuous fields. *IEEE TVCG*, 9(2):206–212, 2003.

[283] J. J. van Wijk and H. van de Wetering. Cushion treemaps: Visualization of hierarchical information. In *Proc. IEEE InfoVis*, pages 73–78, Los Alamitos, CA, 1999. IEEE Press.

[284] C. J. Veenman, M. J. T. Reinders, and E. Backer. Resolving motion correspondence for densely moving points. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(1):54–72, Jan 2001. DOI 10.1109/34.899946.

[285] I. Viola and E. Gröller. Smart Visibility in Visualization. In *Proc. CAe*, pages 209–216, Goslar, Germany, 2005. Eurographics Association. DOI http://dx.doi.org/10.2312/COMPAESTH/COMPAESTH05/209-216.

[286] R. Vliegen, J. J. van Wijk, and E. J. van der Linden. Visualizing business data with generalized treemaps. *IEEE TVCG*, 12(5):789–796, 2006.

[287] L. Voinea. The cvsgrab software evolution visualization system, 2012. URL `http://www.cs.rug.nl/svcg/SoftVis/VCN`.

[288] M. Wan, F. Dachille, and A. Kaufman. Distance-field based skeletons for virtual navigation. In *Proc. IEEE Visualization*, pages 239–246, 2001.

[289] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: From error visibility to structural similarity. *IEEE TIP*, 13 (4):600–612, 2004.

[290] M. Westberg. Time of flight based teat detection. In *Tech. Report LiTH-ISY-EX-09/4154-SE, Univ. of Linköping, Sweden*, 2009. `liu.diva-portal.org/smash/get/diva2:224321/FULLTEXT01.pdf`.

[291] C. F. Westin, S. Peled, H. Gubjartsson, R. Kikinis, and F. A. Jolesz. Geometrical diffusion measures for mri from tensor basis analysis. In *Proc. 5th Annual Meeting of the ISMRM*, page 1742, Berkeley, CA, 1997. International Society for Magnetic Resonance in Medicine.

[292] J. J. van Wijk. Image based flow visualization. *Proc. SIGGRAPH '02, Transactions on Graphics*, 21(3):745–754, 2002.

[293] L. Wilkinson, A. Anand, and R. L. Grossman. Graph-theoretic scagnostics. In *Proc. IEEE InfoVis*, volume 5, page 21, 2005.

[294] N. Willems, H. van de Wetering, and J. J. van Wijk. Visualization of vessel movements. *CGF*, 28(3):959–966, 2010.

[295] G. Winkenbach and D. H. Salesin. Computer-Generated Pen-and-Ink Illustration. In *Proc. SIGGRAPH*, pages 91–100, New York, 1994. ACM. DOI http://doi.acm.org/10.1145/192161.192184.

[296] J. H. Won, M. H. Lee, and I. K. Park. Active 3D shape acquisition using smartphones. In *Proc. IEEE CVPR*, 2012.

[297] P. C. Wong and J. Thomas. Visual analytics. *IEEE Computer Graphics and Applications*, 24(5):20–21, 2004.

[298] C. R. Wren, A. Azarbayejani, T. Darrell, and A. P. Pentland. Pfinder: real-time tracking of the human body. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(7):780–785, Jul 1997. DOI 10.1109/34.598236.

[299] Z. Wu and R. Leahy. An optimal graph theoretic approach to data clustering: theory and its application to image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(11): 1101–1113, Nov 1993. DOI 10.1109/34.244673.

[300] G. X, D. Zhan, and Z. Zhou. Supervised nonlinear dimensionality reduction for visualization and classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 35(6):1098–1107, 2005.

[301] J. Xie, P. Heng, and M. Shah. Shape matching and modeling using skeletal context. *Patt Recog*, 41:1756–1767, 2008.

[302] C. Xu and J. Prince. Gradient vector flow: A new external force for snakes. In *Proc. IEEE CVPR*, pages 66–71, Los Alamitos, CA:, 1997. IEEE Press.

[303] Y. Yang, J. Chen, and M. Beheshti. Nonlinear Perspective Projections and Magic Lenses: 3D View Deformation. *IEEE Computer Graphics & Applications*, 25(1):567–582, January/February 2005. DOI http://dx.doi.org/10.1109/MCG.2005.29.

[304] K. Yau, J. Qadir, H. Khoo, M. Ling, and P. Komisarczuk. A survey on reinforcement learning models and algorithms for traffic signal control. *ACM Computing Surveys*, 50(3), 2017.

[305] A. Yilmaz, X. Li, and M. Shah. Contour-based object tracking with occlusion handling in video acquired using mobile cameras. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(11):1531–1536, Nov 2004. DOI 10.1109/TPAMI.2004.96.

[306] A. Yilmaz, O. Javed, and M. Shah. Object tracking: A survey. *ACM Comput. Surv.*, 38(4), December 2006. DOI 10.1145/1177352.1177355. URL `http://doi.acm.org/10.1145/1177352.1177355`.

[307] A. Yuille, P. Hallinan, and D. Cohen. Feature extraction from faces using deformable templates. *IJCV*, 8(2):99–111, 1992. ISSN 0920-5691. URL `http://dx.doi.org/10.1007/BF00127169`.

[308] H. Zhang, J. E. Fritts, and S. A. Goldman. Image segmentation evaluation: A survey of unsupervised methods. *Computer Vision and Image Understanding*, 100(2):260–280, 2008.

[309] Y. J. Zhang. A survey on evaluation methods for image segmentation. *Pattern Recognition*, 29(8):1335–1346, 1996.

[310] H. Zhou, P. Xu, Y. Xiaoru, and Q. Huamin. Edge bundling in information visualization. *Tsinghua Sci. Tech.*, 18(2):148–156, 2013.

[311] S. C. Zhu, T. S. Lee, and A. L. Yuille. Region competition: unifying snakes, region growing, energy/Bayes/MDL for multi-band image segmentation. In *Proc. IEEE ICCV*, pages 416–423, 1995.

[312] M. Zinsmaier, U. Brandes, O. Deussen, and H. Strobelt. Interactive level-of-detail rendering of large graphs. *IEEE TVCG*, 18(12):2486–2495, 2012.

[313] M. van der Zwan and A. Telea. Robust and fast teat detection and tracking in low-resolution videos for automatic milking devices. In J. Braz, S. Battiato, and F. Imai, editors, *Proceedings of the* 10^{th} *IEEE International Conference on Computer Vision Theory and Applications (VISAPP)*, volume 3, pages 654–667, 2015.

[314] M. van der Zwan, Y. Meiburg, and A. Telea. A dense medial descriptor for image analysis. In J. Braz, S. Battiato, and F. Imai, editors, *Proc.* 8^{th} *IEEE International Conference on Computer Vision Theory and Applications (VISIGRAPP)*, volume 1, pages 133–140, 2013.

[315] M. van der Zwan, V. Codreanu, and A. Telea. CUBu: Universal real-time bundling for large graphs. *IEEE TVCG*, 22(12):2250–2263, 2016.

[316] M. van der Zwan, W. Lueks, H. Bekker, and T. Isenberg. Illustrative Molecular Visualization with Continuous Abstraction. *Computer Graphics Forum*, 30(3):683–690, May 2011.

[317] M. van der Zwan, A. Telea, and T. Isenberg. Continuous navigation of nested abstraction levels. In M. Meyer and T. Weinkauf, editors, *Proc. EG/IEEE VGTC Conference on Visualization (EuroVis) – Short papers*, pages 13–17, 2012.

[318] M. van der Zwan, A. Telea, and T. Isenberg. Smooth navigation between nested spatial representations. In *Proceedings of the National ICT.OPEN/SIREN 2012 Workshop, October 22–23, 2012, Rotterdam, The Netherlands*, pages 140–144, 2012.

ABOUT THE AUTHOR

Matthew Anthony Thomas van der Zwan was born on the 11th of September of 1987 in Groningen, The Netherlands. He received his bachelor degrees in Computer Science and Mathematics from the University of Groningen in 2009.

His interest in visualization and computer graphics in general piqued, he went on to pursue a masters's degree at the University of Groningen in the area of computational science and visualization. The research performed in this time focused on bringing together visualization techniques with different levels of abstraction. For an internal research internship, he worked on integrating different molecular visualization techniques. Following that, he focused on fluid flow visualizations for his master's thesis. He succesfully defended his master's thesis and received his master's degree cum laude in 2011.

The research on molecular visualization performed during his master's led to his first publication. Enjoying doing research, he decided to accept the offer to become a PhD student at the University of Groningen in 2011, this time in the area of both visualization and computer vision.

The motivation for this PhD thesis comes from realizing, during his time as a PhD student, that a lot of the data produced from, for example, computer vision and used for visualization are of a similar form and can be grasped under one term: multidimensional time-dependent trails. Therefore, a lot of processing and visualization techniques (such as bundling) are applicable in a much broader domain than they currently are even though the final visualization might be very application specific. This thesis provides the results of the investigation into multiple aspects of acquiring and visualizing multidimensional time-dependent trails.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

`http://code.google.com/p/classicthesis/`

Final Version as of July 22, 2018 (`classicthesis`).