TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

DEPARTMENT OF COMPUTER SCIENCE
VISUAL INFORMATION ANALYSIS GROUP

Topology-Based Characterization and Visual Analysis of Feature Evolution in Large-Scale Simulations

Thesis approved by the Department of Computer Science
of the Technische Universität Kaiserslautern
for the award of the Doctoral Degree
Doctor of Natural Sciences (Dr. rer. nat.)

to

Jonas Lukasczyk

Date of Defense:	19 July 2019
Dean:	Prof. Dr. Stefan Deßloch
Reviewers:	Prof. Dr. Heike Leitte, TU Kaiserslautern Prof. Dr. Gunther H. Weber, Lawrence Berkeley National Laboratory Dr. Julien Tierny, CNRS - Sorbonne Université

D 386

ABSTRACT

This manuscript presents a topology-based analysis and visualization framework that enables the effective exploration of feature evolution in large-scale simulations. Such simulations pose additional challenges to the already complex task of feature tracking and visualization, since the vast number of features and the size of the simulation data make it infeasible to naively identify, track, analyze, render, store, and interact with data. The presented methodology addresses these issues via three core contributions. First, the manuscript defines a novel topological abstraction, called the Nested Tracking Graph (NTG), that records the temporal evolution of features that exhibit a nesting hierarchy, such as superlevel set components for multiple levels, or filtered features across multiple thresholds. In contrast to common tracking graphs that are only capable of describing feature evolution at one hierarchy level, NTGs effectively summarize their evolution across all hierarchy levels in one compact visualization. The second core contribution is a view-approximation oriented image database generation approach (VOIDGA) that stores, at simulation runtime, a reduced set of feature images. Instead of storing the features themselves—which is often infeasable due to bandwidth constraints—the images of these databases can be used to approximate the depicted features from any view angle within an acceptable visual error, which requires far less disk space and only introduces a neglectable overhead. The final core contribution combines these approaches into a methodology that stores *in situ* the least amount of information necessary to support flexible *post hoc* analysis utilizing NTGs and view approximation techniques.

ACKNOWLEDGEMENTS

The research presented in this dissertation would have not been possible without the outstanding commitment and support of all my supervisors: Prof. Dr. Heike Leitte, Prof. Dr. Christoph Garth, and Prof. Dr. Gunther H. Weber. I experienced the best working atmosphere imaginable, and our discussions have been invaluable to me.

I am extremely thankful that I was part of the international research training group IRTG2057 that provided me with opportunities to attend international conferences, to enhance my inter-cultural competences due to several research stays abroad, and to learn from many distinguished international experts. The international exchange of knowledge was instrumental in my research. Therefore, I thank all my colleagues and collaborators from the TU Kaiserslautern, the Fraunhofer ITWM, the Arizona State University (ASU), and the Los Alamos National Laboratory (LANL). I want to especially thank Prof. Dr. Ross Maciejewski (ASU) for his continuous guidance throughout my entire doctoral studies, and Dr. James Ahrens (LANL) for arranging my visits to the national laboratory. I would also like to express my deep appreciation to Prof. Dr. Hans Hagen who continuously gave me highly valued professional as well as personal advice.

Finally, I thank my friends and family for their constant emotional support. First and foremost, I thank my wife Qiong Xiao for coping with my trips abroad and my long working hours during paper deadlines. I thank my parents—Bettina and Walter Lukasczyk—who are a fixed anchor in my life I can always rely on. Everyone involved in my life made my time as a doctoral student an enriching and beautiful experience.

CONTENTS

LIST OF FIGURES

List of Definitions

List of Algorithms

NOTATIONS

$\mathbb{X}, \mathbb{Y}$	Topological space
$\mathbb{R}^d$	Euclidean space of dimension d
$\mathbb{N}_{<n}$	Set of natural numbers including zero smaller than n
$\mathbb{Z}$	Set of integers
$card(X)$	Cardinality of set X
$X \setminus Y$	Set X without elements of set Y
$\dot{X}$	Connected component of set X
$\bar{X}$	Enumeration of all elements of set X
f^{-1}	Inverse of function f
σ	Simplex
$\langle p_0, ..., p_d \rangle$	A d-simplex for affinely independent points $p_0, ..., p_d$
$\mathcal{K}$	Simplicial complex
$\lvert\mathcal{K}\rvert$	Underlying space of simplicial complex $\mathcal{K}$
$\mathbb{M}$	Manifold
$\mathcal{M}$	Piecewise linear manifold
$\mathcal{L}_f(l)$	Level set of function f for isovalue l
$\mathcal{L}_f^-(l)$	Sublevel set of function f for isovalue l
$\mathcal{L}_f^+(l)$	Superlevel set of function f for isovalue l
$\dot{\mathcal{L}}_{f,x}(l)$	Level set component of $\mathcal{L}_f(l)$ that contains point x
$\mathbb{M}/\sim$	Quotient space of manifold $\mathbb{M}$ under equivalence relation $\sim$
$\mathcal{R}_f$	Reeb graph of function f
$\mathcal{C}_f$	Contour tree of function f
$\mathcal{C}_f^-$	Join tree of function f
$\mathcal{C}_f^+$	Split tree of function f
$\mathcal{C}_f^*$	Join, Split, or Contour Tree of function f
$\mathcal{P}_f$	Persistence diagram of function f
$\mathcal{S}$	Segmentation
$\dot{\mathcal{S}}$	Sublevel and superlevel set component segmentation
$\tilde{\mathcal{S}}$	Contour, join, and split tree segmentation
$\mathcal{T}$	Tracking graph
$\mathcal{N}$	Nested tracking graph

CHAPTER 1

INTRODUCTION

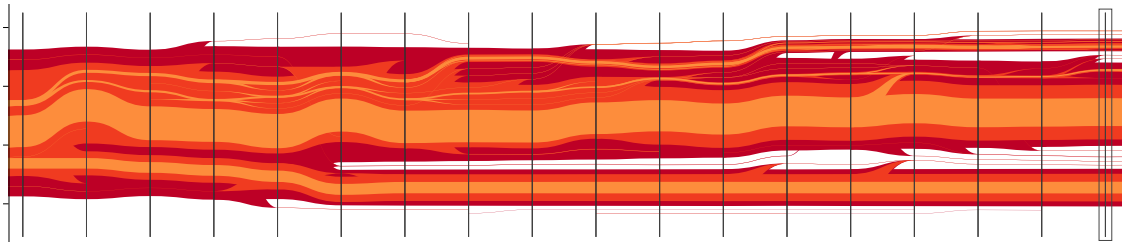
In many scientific domains, interesting features correspond to areas that exceed some threshold. Examples include highly turbulent regions in flow fields (vortices), areas in combustion simulations above a fuel consumption rate threshold (burning cells), parts of the universe exceeding a dark matter density (halos), and regions in a fluid with a minimum viscosity (viscous fingers). Simulating models of these phenomena is crucial to understand them, since simulations provide the means to verify hypothesis, to derive predictions, and—most importantly—to iteratively advance the models. To this end, it is essential for simulation analysis to reliably identify individual features, and to robustly correlate them over time, which enables the characterization of features, as well as their evolution, properties, and mutual interaction. Recent advances in high-performance computing enable large-scale simulations of such models, which introduces additional challenges to the already complex task of simulation analysis, as the vast number of features and the size of the simulation data make it infeasible to naively identify, track, render, store, and interact with features.

1.1 SCOPE

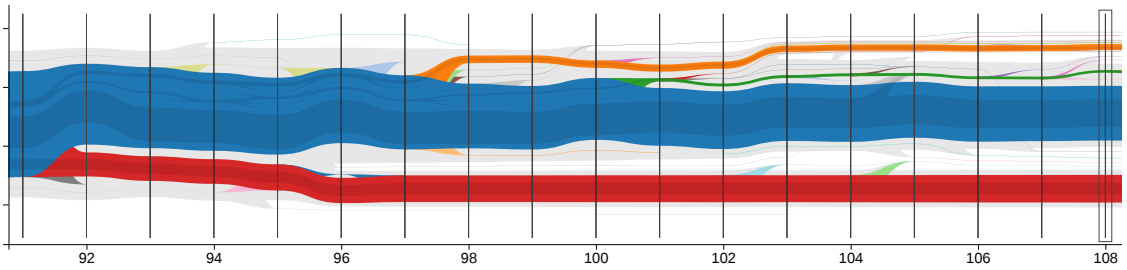
The methodology described in this manuscript focuses on a central limitation of large-scale simulations, i.e., the fact that it is usually infeasible to store every simulation state in its entirety due to bandwidth and disk space constraints [2, 68, 118]. This limitation becomes more apparent as we approach the era of exascale computing. A solution to this problem is to deploy *in situ* analysis algorithms that process simulation data while it is still in machine memory. The purpose of these algorithms is to determine and store, at simulation runtime, the least amount of information necessary to support flexible *post hoc* analysis; including the capability to identify, filter, track, and render features. At the same time, effectively exploring numerous features and their complex evolutions requires visual analytics frameworks that partition features into a manageable amount of groups. This hierarchical decomposition enables the visual analysis of features in a level-of-detail approach, by providing overviews of feature groups, as well as detailed information about individual features, their properties, and evolutions.

1.2 CONTRIBUTIONS

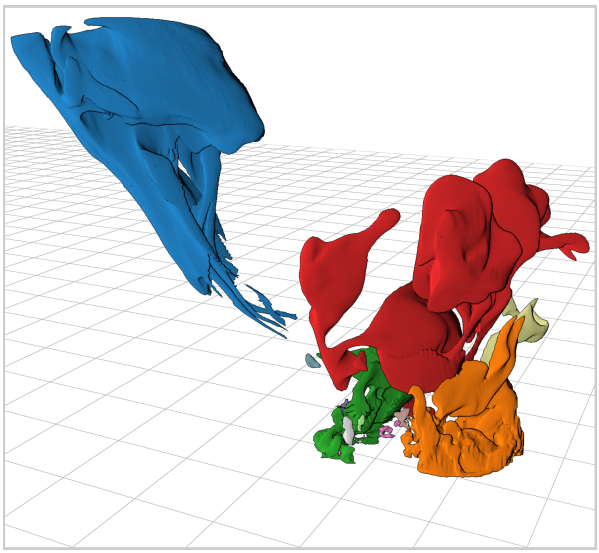
Topological data analysis enables the precise and robust characterization of feature evolution in scalar fields, including abstractions that identify features as domain subsets based on field values (level, sublevel, and superlevel set components), that describe the structure of the entire field (contour trees), that rank and filter features according to their significance (persistence), and that record the temporal evolution of features (tracking graphs). This work presents a novel topological abstraction, called the Nested Tracking Graph (NTG), that records the temporal evolution of features that have a nesting hierarchy; such as superlevel set components for multiple levels, or filtered features across multiple persistence thresholds. In contrast to common tracking graphs that are only capable of describing feature evolution at one hierarchy level, NTGs effectively summarize feature evolution at all hierarchy levels in one compact visualization (Fig. 1.1a-b). Each layer of a NTG is a common tracking graph, where edges of different layers are drawn inside each other based on the nesting hierarchy of the features. This hierarchy is recorded for each timestep by a so-called merge tree (Fig. 1.1d). The proposed methodology stores, at simulation runtime, the merge trees and other intermediate data structures to efficiently compute NTGs for any feature parameters *post hoc*, without requiring access to the original simulation data. NTGs then enable analysts to effectively peel through the simulation data by interactively updating feature parameters, and following the history of individual features and feature groups (Fig. 1.1b-d).



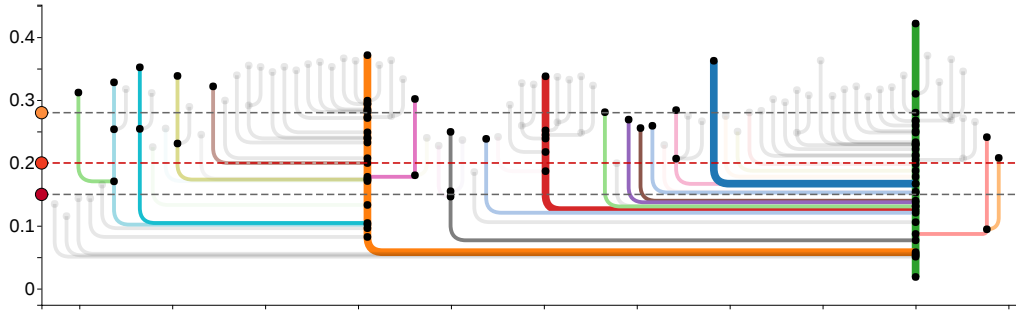
(a) Nested tracking graph colored by layer



(b) Nested tracking graph colored by individual features of the second layer



(c) 3D view composed of individual feature images



(d) Merge tree illustrating the significance and nesting hierarchy of features

Figure 1.1: Individual elements of the proposed methodology demonstrated on the asteroid impact case study described in Sec. 5.4.2.

To render feature geometries *post hoc*, the proposed methodology uses a view-approximation oriented image database generation approach (VOIDGA) to store, at simulation runtime, a collection of feature images for a predefined set of parameter values (Fig. 1.1c). Instead of storing the feature geometries themselves—which is often infeasable due to bandwidth constraints—VOIDGA stores a reduces set of images that can be used by image-based rendering techniques to approximate the depicted geometries from any view angle with an acceptable visual error, which requires far less disk space and only introduces a neglectable compositing overhead.

Finally, the proposed methodology combines NTGs and image databases to derive *in situ* a database that enables flexible *post hoc* analysis within a topology-based visual analytics framework. The core interaction device of this framework is a NTG that is dynamically computed based on analysis products stored in the database, which is used to browse through time, update feature parameters, aggregate features into groups, and to retrieve feature images and other data products form the database. This approach enables the efficient interactive analysis of large-scale simulations based on a relatively small database. The primary advantage of this methodology is that the database only grows proportional to a predefined parameter sampling—e.g., based on a maximum number of images, or a list of level values—independent of the actual size of the simulation data and the number of features. This decoupling enables the approach to scale to state-of-the-art, largest-scale simulations which stand to benefit from the proposed methodology.

1.3 STRUCTURE

This manuscript is structured as follows.

Ch. 2 introduces the background of the proposed methodology, including the data representation, feature characterization, tracking approaches, and view-approximation techniques.

Ch. 3 then uses these definitions to formalize NTGs and demonstrate their effectiveness in several case studies.

Ch. 4 describes the view-approximation oriented image database generation approach (VOIDGA) that derives, at simulation runtime, a reduced set of feature images that are composited during *post hoc* analysis to render 3D scenes.

Ch. 5 combines NTGs and image databases to derive *in situ* a database that enables flexible *post hoc* analysis.

Ch. 6 summarizes the results of the proposed methodology and provides an outlook on future research directions.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter introduces the topological concepts and data structures which are the basis of this work. This includes state-of-the-art approaches for feature characterization, tracking, scientific image databases, and view approximation techniques. Specifically, the first section (Sec. 2.1) build up to a formal definition of the primary data representation that is used throughout this manuscript (Sec. 2.2). This representation enables the formalization of topology-based feature characterizations (Sec. 2.3), which are then used to formalize tracking techniques for these features (Sec. 2.4). The second contribution of this work is an interactive interface design for large-scale simulations that enables the real-time visualization of features. The core components of this interface are scientific image databases (Sec. 2.5), and view approximation techniques (Sec. 2.6).

2.1 PRELIMINARY DEFINITIONS

This subsection contains preliminary definitions that are used throughout this manuscript to formalize the data representation and feature characterization.

Definition 1 (Topology) *A topology on a set X is a collection T of subsets of X having the following properties:*

(i) The sets $\emptyset$ and X are in T;

(ii) The union of any sub-collection of T is in T, and;

(iii) The intersection of a finite sub-collection of T is in T.

Definition 2 (Topological Space) *A topological space $\mathbb{X} = (X, T)$ is a pair of a set X and a topology T defined on X.*

Definition 3 (Function) *A function $f : \mathbb{X} \to \mathbb{Y}$ associates each element of the topological space $\mathbb{X}$ with exactly one element of the topological space $\mathbb{Y}$. The inverse of f is denoted by $f^{-1} : \mathbb{Y} \to \mathbb{X}$.*

Definition 4 (Injection) *A function $f : \mathbb{X} \to \mathbb{Y}$ is an injection iff*

$$\forall x_1, x_2 \in \mathbb{X} \ : \ x_1 \neq x_2 \Rightarrow f(x_1) \neq (x_2).$$

Definition 5 (Surjection) *A function $f : \mathbb{X} \to \mathbb{Y}$ is a surjection iff*

$$\forall y \in \mathbb{Y} \ : \ \exists x \in \mathbb{X} \ s.t. \ f(x) = y.$$

Definition 6 (Bijection) *A function f is a bijection iff it is an injection and a surjection.*

Definition 7 (Enumeration) *An enumeration is a bijection $\bar{X} : \mathbb{N}_{<n} \to X$ that maps the first n natural numbers to the elements of a collection X where $n = card(X)$. Hence, $\bar{X}^{-1}(x)$ returns the index of element $x \in X$ in $\bar{X}$. To simplify notations, an enumeration can also be denoted as a sequence*

$$\bar{X} = (\bar{X}_0, \bar{X}_1, \ldots, \bar{X}_{n-1}).$$

Definition 8 (Open Set) *A set $X \subseteq \mathbb{X}$ of a topological space $\mathbb{X}$ is an open set of $\mathbb{X}$ if it is in the topology T of $\mathbb{X}$.*

Definition 9 (Closed Set) *A set $X \subseteq \mathbb{X}$ of a topological space $\mathbb{X}$ is a closed set of $\mathbb{X}$ if its compliment $\mathbb{X} \backslash X$ is open.*

Definition 10 (Continuous Function) *A function $f : \mathbb{X} \to \mathbb{Y}$ is continuous iff for each open subset $Y \subseteq \mathbb{Y}$ the set $f^{-1}(Y)$ is an open subset of $\mathbb{X}$.*

Definition 11 (Homeomorphism) *A homeomorphism between two topological spaces $\mathbb{X}$ and $\mathbb{Y}$ is a continuous bijection $f : \mathbb{X} \to \mathbb{Y}$ whose inverse $f : \mathbb{Y} \to \mathbb{X}$ is also continuous.*

Definition 12 (Homeomorphic) *Two topological spaces are said to be homeomorphic iff there exists a homeomorphism between the two spaces.*

Definition 13 (Homotopy) *A homotopy between two continuous functions $f, g : \mathbb{X} \to \mathbb{Y}$ is a continuous function $h : \mathbb{X} \times [0,1] \to \mathbb{Y}$ such that $h(x,0) = f(x)$ and $h(x,1) = g(x)$ for all $x \in \mathbb{X}$.*

Definition 14 (Homotopic) *Two continuous functions f and g are said to be homotopic iff there exists a homotopy between the two functions.*

Definition 15 (Path) *A homeomorphism $p : (0,1) \to Y$ from the unit interval to a subset $Y \subseteq \mathbb{Y}$ is called a path between the points $p(0)$ and $p(1)$ in the topological space $\mathbb{Y}$.*

Definition 16 (Connected Topological Space) *A topological space $\mathbb{X}$ is connected iff there exists a path on $\mathbb{X}$ between any two distinct points of $\mathbb{X}$.*

Definition 17 (Connected Component) *A connected component $\dot{X}$ (sometimes just referred to as a component) is a maximal connected subset of a set X.*

Definition 18 (Simply Connected) *A topological space $\mathbb{X}$ is simply connected iff it is connected and any two paths on $\mathbb{X}$ between any two distinct points of $\mathbb{X}$ are homotopic.*

Note, if there exists a homeomorphism between two topological spaces, then any connected neighborhood of one space can be mapped to a connected neighborhood of the other, and vice versa. If these neighborhoods can even be deformed into each other by a continuous function, then this deformation function is called a homotopy.

2.2 DATA REPRESENTATION

Many scientific simulations and experiments of physical phenomena are modeled as collections of real-valued functions $f : \mathbb{X} \to \mathbb{R}$ over a topological space $\mathbb{X}$. However, the values of f are often only computed or observed at a finite set of points. This section describes how to represent the underlying space with a finite collection of simple pieces that connect these points (Sec. 2.2.1), and how values only available at these points can be interpolated across the entire space of the representation (Sec. 2.2.2).

2.2.1 Domain Representation

The methodology described in this manuscript processes representations of topological spaces based on collections of simple pieces (called simplices) that constitute more complex structures (called simplicial complexes) [25].

Simplicies

Formally, a d-simplex is the convex hull of $d+1$ affinely independent points in $\mathbb{R}^n$ with $0 \leq d \leq n$, where the most basic simplex corresponds to a single point (called a vertex). Vertices play a special role as they (a) represent the discrete locations on where data is available, and (b) define higher dimensional simplices. The second property enables the description of the space between vertices via a piecewise linear combination of the vertices (Fig. 2.1), which is convenient to interpolate values within simplices (Sec. 2.2.2).

Definition 19 (Convex Set) *A subset $X \subset \mathbb{R}^n$ is convex if for any two points $x_0, x_1 \in X$ and $\lambda \in [0,1]$ the point $p = (1-\lambda)x_0 + \lambda x_1$ is also an element of X.*

Definition 20 (Convex Hull) *The convex hull of a point set $X \subset \mathbb{R}^n$ is the unique minimal convex set containing all points of X.*

Definition 21 (Simplex) *A d-simplex $\sigma = \langle p_0, ..., p_d \rangle$ is the convex hull of $d+1$ affinely independent points $p_i \in \mathbb{R}^n$ with $0 \leq d \leq n$. The points p_i are called the generators of σ.*

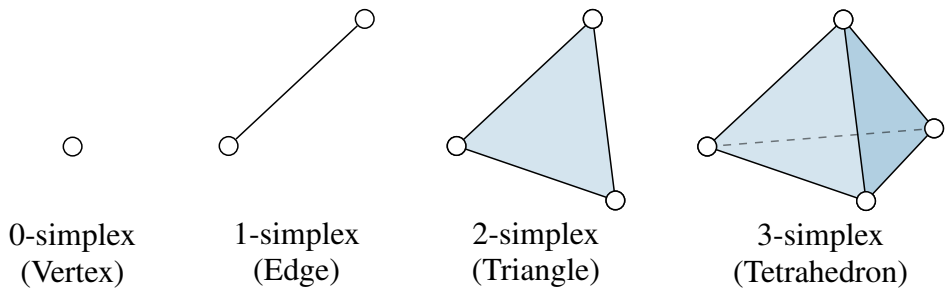


Figure 2.1: Illustration of d-simplices for $0 \leq d \leq 3$.

Simplicial Complexes

The next step is to "glue" multiple simplices together to build more complex structures, called simplicial complexes (Fig. 2.2). This requires the definition of a simplex face, i.e., the areas of simplices on which it is allowed to glue them together. Hence, a face of a d-simplex σ is a simplex that is defined by any non-empty subset of the $d+1$ vertices of σ. For instance, the faces of a triangle (a 2-simplex) are its vertices (0-simplices), edges (1-simplices), and the triangle itself. Moreover, the boundary of a d-simplex is the union of all its d-1-faces, e.g., the boundary of a triangle is the union of its edges. Based on these definitions, a simplicial complex corresponds to a collection of simplices together with all their faces, where two simplices are either disjunct or their intersection is a complete face of both simplices. This ensures that simplices are only connected via a common face; which prohibits degenerate structures.

Definition 22 (Simplex Face) *A face τ of a d-simplex σ is any simplex defined by a non-empty subset of the $d+1$ generators of σ, and is denoted by $\tau \leq \sigma$.*

Definition 23 (Simplex Boundary) *The boundary of a d-simplex σ is the union of its d-1 dimensional faces.*

Definition 24 (Simplex Interior) *The interior of a d-simplex σ is the set of points of σ that are not elements of its boundary.*

Definition 25 (Simplicial Complex) *A simplicial complex $\mathcal{K}$ is a finite collection of non-empty simplices such that (i) every face of a simplex $\sigma \in \mathcal{K}$ is also in $\mathcal{K}$, and (ii) any two simplices of $\mathcal{K}$ intersect in a common face or not at all.*

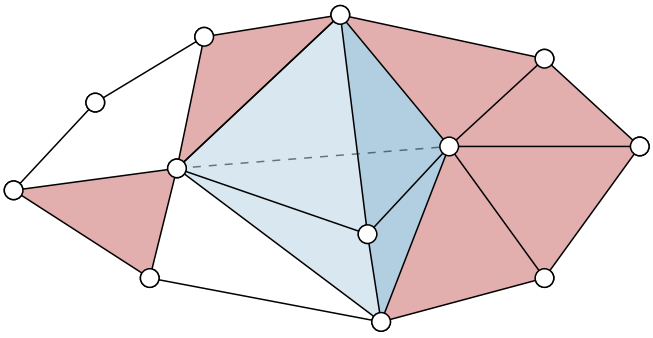


Figure 2.2: Illustration of a simplicial complex consisting of vertices (discs), edges (lines), triangles (red surfaces), and tetrahedra (blue volumes). Note, simplicies only intersect at a common face or not at all.

Local Neighborhoods

Topological algorithms—such as the contour tree algorithm by Carr et al. [13]—utilize the connectivity of simplicial complexes. Two important subsets of complexes that are essential for such algorithms are the so-called *star* and *link* of a simplex. Both subsets describe the local neighborhood of a simplex within a complex in any dimension. Specifically, the star of a simplex is the subset of simplices that have the simplex as a face (Fig. 2.3, first row). Note, a star is not necessarily a simplicial complex as it might violate condition (i) of Def. 25. The union of the star with all its faces yields a simplicial complex called the closed star.

The link of a simplex σ is defined as the subset of simplices of the closed star that are disjoint from σ (Fig. 2.3, second row). The link can be interpreted as the set of simplices in the local neighborhood of σ that are transitively connected through σ. For a function that assigns a real number to the vertices of a simplicial complex, the link can again be partitioned into two subsets: the lower and upper link. The upper link of a simplex contains only simplices whose vertices have a strictly larger value then the ones of the simplex, whereas the lower link only contains simplices whose vertices have a strictly smaller value (Fig. 2.3, third row).

As described in Sec. 2.3, vertex stars and links can be used to efficiently, and independently identify critical points on simplicial complexes—e.g., minima and maxima—since these subsets represent the local topological neighborhood of their respective vertices.

Definition 26 (Star) *The star of a simplex $\tau \in \mathcal{K}$ is the set of all simplices of a simplicial complex $\mathcal{K}$ that contain τ, i.e., $St(\tau) = \{\sigma \in \mathcal{K} \mid \tau \leq \sigma\}$.*

Definition 27 (Closed Star) *The closed star $\bar{St}(\sigma)$ of a simplex σ is the union of $St(\sigma)$ and all its faces $\tau \leq St(\sigma)$.*

Definition 28 (Link) *The link of a simplex $\sigma \in \mathcal{K}$ is the set of faces $\tau \in \bar{St}(\sigma)$ that have an empty intersection with σ, i.e., $Lk(\sigma) = \{\tau \in \bar{St}(\sigma) \mid \tau \cap \sigma = \emptyset\}$.*

Definition 29 (Lower/Upper Link) *For a function f that assigns a real number to the vertices of a simplicial complex $\mathcal{K}$, the lower link $Lk^-(\sigma)$ (respectively upper link $Lk^+(\sigma)$) of a simplex $\sigma \in \mathcal{K}$ are the simplices of $Lk(\sigma)$ whoes vertices all have a value stricly lower (respectively larger) than the ones of σ.*

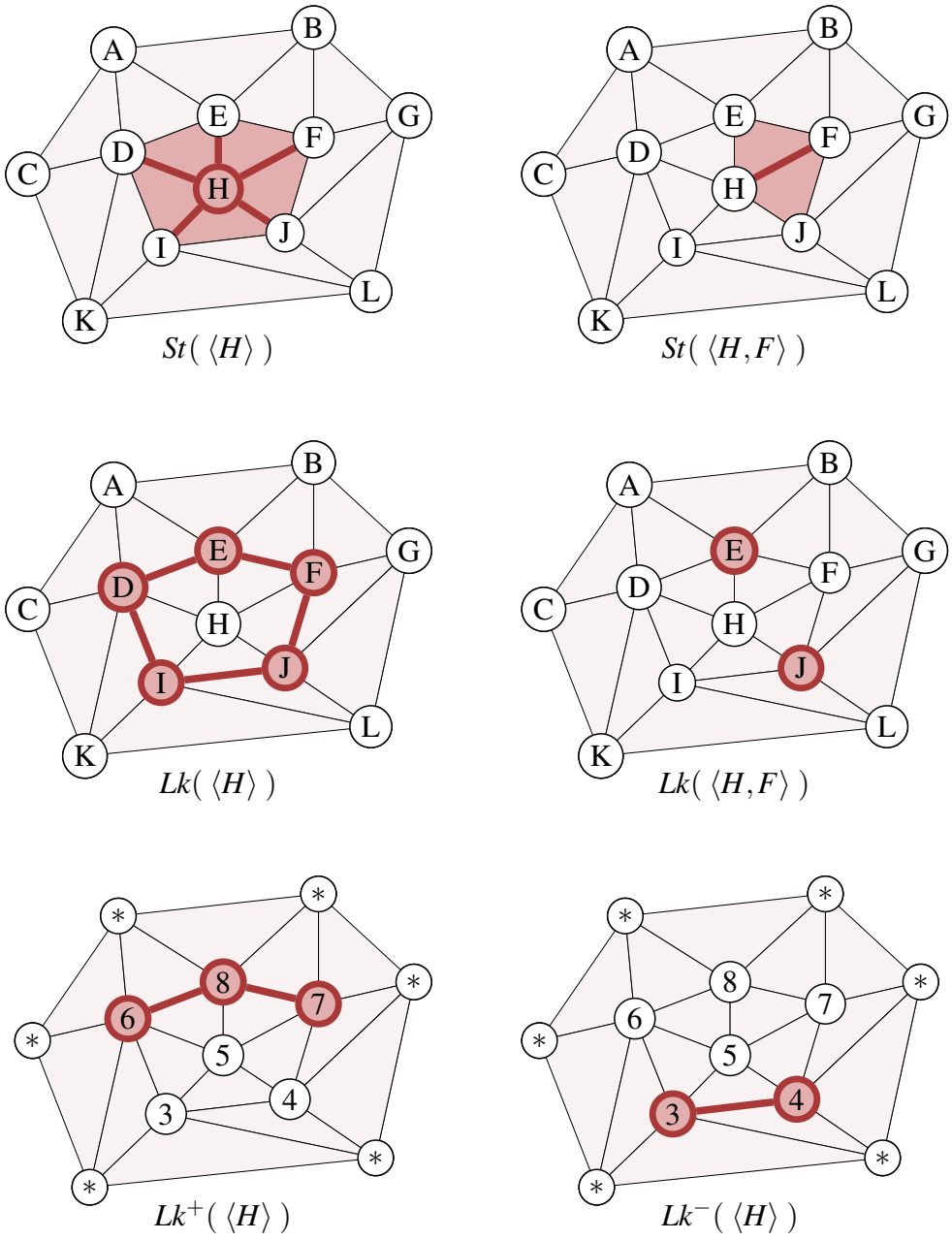


Figure 2.3: Examples that show the *Star* (first row) and *Link* (second row) of a vertex $\langle H \rangle$ and an edge $\langle H, F \rangle$, as well as the *Upper Link* (bottom left) and *Lower Link* (bottom right) of vertex $\langle H \rangle$.

Triangulations of Manifolds

As described previously, a simplicial complex enables the representation of topological spaces via collections of simple pieces. However, the question arises under which conditions a simplicial complex approximates a topological space well, and if such a representation even exists. In order to compare a topological space $\mathbb{X}$ to a simplicial complex $\mathcal{K}$, consider the union of its simplices $|\mathcal{K}| = \bigcup \mathcal{K}$, which is referred to as the underlying space of the complex. Then, $\mathcal{K}$ is said to be topologically equivalent to $\mathbb{X}$ iff its underlying space is homeomorphic to $\mathbb{X}$. In this case, the complex $\mathcal{K}$ is called a triangulation of the space $\mathbb{X}$.

This work focuses on a specific subset of topological spaces called d-manifolds, which locally correspond to the Euclidean space $\mathbb{R}^d$. Thus, every interior point of a d-manifold has an open neighborhood that can be continuously transformed via a homeomorphism to a d-dimensional open Euclidean ball, and every point on its boundary into a half-ball. Predominant examples for such spaces are surfaces and volumes, which are 2-manifolds and 3-manifolds, respectively. To be more precise, the methods proposed in this work focus on simply connected d-manifolds, i.e., manifolds without holes, voids, and so forth. As described in Sec. 2.3.2, this restriction is required for some topological concepts such as the contour tree.

If a simplicial complex is a triangulation of a manifold—i.e., if their respective spaces are homeomorphic—then the triangulation is also called a piecewise linear manifold. Fig. 2.4 illustrates an example of a topologically equivalent and an inequivalent representation of a 2-manifold. The term piecewise linear will become clear in the next section, which describes how data can be linearly interpolated within the complex.

Definition 30 (Underlying Space) *The underlying space $|\mathcal{K}|$ of a set of simplicies $\mathcal{K}$ is the union of the simplices.*

Definition 31 (Triangulation of a Topological Space) *A triangulation of a topological space $\mathbb{X}$ is a simplicial complex whose underlying space is homeomorphic to $\mathbb{X}$.*

Definition 32 (Manifold) *A topological space $\mathbb{M}$ is a d-manifold if every interior point (respectively boundary point) $x \in \mathbb{M}$ has an open neighborhood that is homeomorphic to an open Euclidean ball (respectively half-ball) of dimension d.*

Definition 33 (Piecewise Linear Manifold) *A piecewise linear manifold $\mathcal{M}$ is a triangulation of a manifold $\mathbb{M}$.*

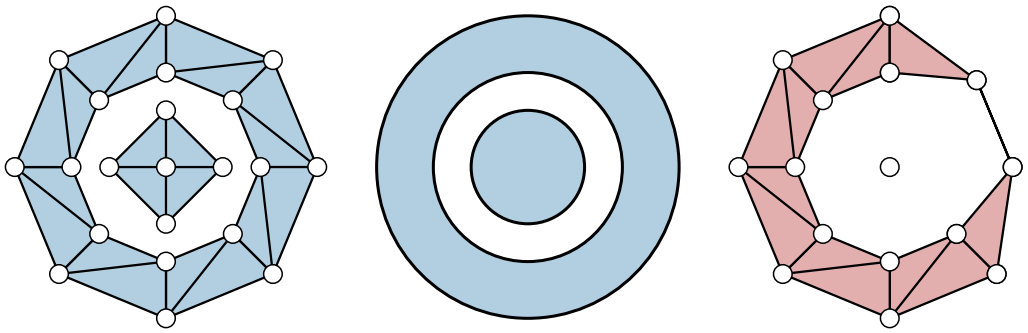


Figure 2.4: Examples of a topologically equivalent simplicial complex (left) and an inequivalent complex (right) triangulation of a 2-manifold $\mathcal{M}$ consisting of two connected components (middle). The simplicial complex on the right is not a triangulation of $\mathcal{M}$ as its underlying space is not homeomorphic to $\mathcal{M}$. Specifically, the single vertex in the center is not homeomorphic to the disc as any function that maps open sets of the center component of $\mathcal{M}$ to a single point is not injective. The same argument applies for the line segment of the outer complex and the corresponding part of the outer component of $\mathcal{M}$.

Simulations of physical phenomena often compute values on the vertices of a pieces linear manifold. Alternatively, vertices can also be connected by other elementary pieces instead of simplicies, such as cubes or pyramids. These pieces, however, can be decomposed into a set of simplicies, e.g., a cube can be represented by five tetrahedra. Thus, piecewise linear manifolds are the prime domain representation for the remainder of this manuscript, since they cover a wide range of common simulation setups.

2.2.2 Value Representation

The simulations considered in this manuscript are based on a model of a function $f : \mathbb{X} \to \mathbb{R}$ that assigns to each point of a topological space $\mathbb{X}$ a real number. Each such function is referred to as a scalar field. The concrete implementation of a simulation discretizes the topological space $\mathbb{X}$ via a piecewise linear (PL) manifold $\mathcal{M}$, and then computes a real-valued function $\mathring{f} : \mathcal{V} \to \mathbb{R}$ on the vertices $\mathcal{V} \subseteq \mathcal{M}$. Values inside a higher dimensional simplex $\sigma \in \mathcal{M}$ can be computed by a linear combination of the vertex values of σ. Common weights for such a linear combination are the barycentric coordinates of a point inside σ, which correspond to the fractions of hypervolumes [41]. A scalar field that in this way linearly interpolates the values inside a PL manifold is called a PL scalar field (Fig. 2.5). These functions have two important properties: (i) they can be computed very efficiently, and (ii) their gradient is piecewise constant.

The methodology described in this manuscript processes an enumeration of PL scalar fields that are defined on the same PL manifold. In the context of simulations, each element of the enumeration corresponds to an individual state, and the order represents time. Such an enumeration is called a time-varying PL scalar field.

Definition 34 (Scalar Field) *A scalar field is a function $f : \mathbb{X} \to \mathbb{R}$ that assigns to each point of a topological space $\mathbb{X}$ a real number.*

Definition 35 (Barycentric Coordinates) *For a d-simplex $\sigma = \langle v_0, v_1, \ldots, v_d \rangle$ and a point $p \in \sigma$, the uniquely determined real coefficients $(\alpha_0, \alpha_1, \ldots, \alpha_d)$ for which*

$$p = \sum_{i=0}^{d} \alpha_i v_i \quad \textit{and} \quad 1 = \sum_{i=0}^{d} \alpha_i$$

are called the barycentric coordinates of p relative to σ.

Definition 36 (Piecewise Linear Scalar Field) *Let $\mathring{f} : \mathcal{V} \to \mathbb{R}$ be a scalar field that assigns to the vertices $\mathcal{V}$ of a PL manifold $\mathcal{M}$ a real number. Then the value of any point p inside a d-simplex $\sigma = \langle v_0, v_1, \ldots, v_d \rangle \in \mathcal{M}$ can be interpolated via the piecewise linear scalar field $\hat{f} : |\mathcal{M}| \to \mathbb{R}$ defined as*

$$\hat{f}(p) = \sum_{i=0}^{d} \alpha_i \mathring{f}(v_i)$$

where the coefficients α_i are the barycentric coordinates of p relative to σ.

Definition 37 (Time-Varying Piecewise Linear Scalar Field) *A time-varying PL scalar field is an enumeration of PL scalar fields on the same PL manifold ordered by time.*

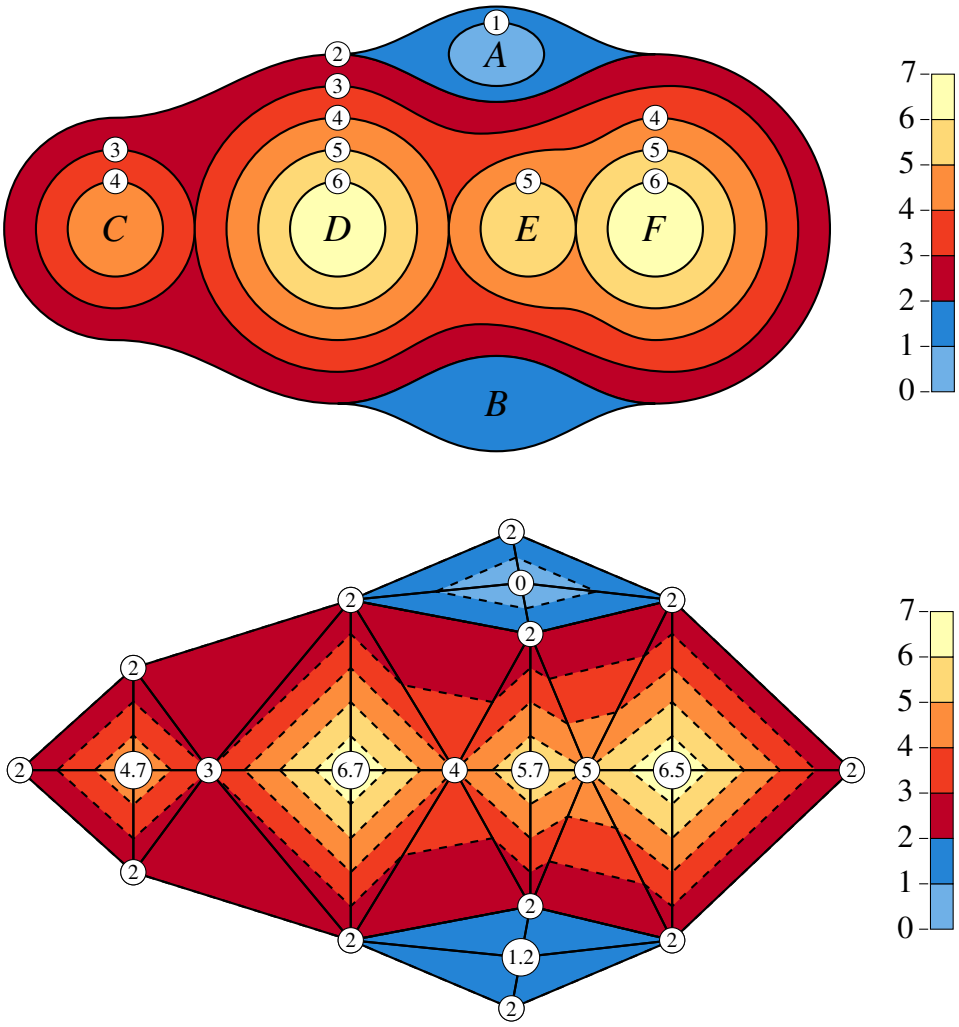


Figure 2.5: Piecewise linear scalar field representation (bottom) of a scalar field (top). Top: Scalar field f over manifold $\mathbb{M}$ with local extrema (A: 0), (B: 1.2), (C: 4.7), (D: 6.7), (E: 5.7), and (F: 6.5). Colors encode integer intervals for values from light blue over dark red to bright yellow. Borders between intervals—also referred to as contours (Sec. 2.3.1)—are illustrated by solid lines whose values are denoted by attached white discs. Bottom: PL scalar field $\hat{f}$ defined on a PL manifold $\mathcal{M}$ of $\mathbb{M}$ where values at vertices (white discs) are probed using f. Note, contours (dashed lines) of $\hat{f}$ for values 1 to 6 can be linearly interpolated on the simplicies of $\mathcal{M}$ (vertices, edges, and triangles).

2.3 TOPOLOGY-BASED FEATURE CHARACTERIZATION

This section builds up to a rigorous topological feature characterization for piecewise linear scalar fields based on contour trees [108], starting from simple function value-based segmentation to hierarchical feature decomposition. These concepts are described in detail since they are fundamental to the proposed methodology. Specifically, the primary feature characterizations used in this work are *sublevel*, *level*, and *superlevel sets*, which correspond to domain subsets for which function values are either smaller than, equal to, or larger than a predefined value, respectively. Essential structures of these sets are so-called *contours*, which correspond to connected components of level sets, and are the boundaries of sublevel and superlevel sets. In brief, the proposed approach segments, simplifies, and extracts domain subsets based on these contours and their evolution, which are represented by a topological data structure called the *contour tree* [108].

An outline of the entire characterization procedure is shown in Fig. 2.6. In various applications, important regions in scalar fields (Fig. 2.6a left) can be characterized via level, sublevel, and superlevel sets for certain values (Sec. 2.3.1). However, these sets are very sensitive to that value, since they might drastically change in shape and number even for small value variations. This problem leads to the study of topological changes of these sets while they evolve during a continuous value sweep (Sec. 2.3.2). The evolution of contours during the sweep are represented by the contour tree [108], where edges correspond to individual contours, and nodes indicate when contours appear, merge, split, and disappear (Fig. 2.6a right). The sweep also partitions the domain into regions, called segments, that correspond to branches of the contour tree (Fig. 2.6b). A key result of this line of research [87] is that contours only change topologically at so-called critical points (Sec. 2.3.3). This fact makes it possible to efficiently compute the contour tree and the corresponding domain segmentation in the piecewise linear setting (Sec. 2.3.4). In general, however, datasets may exhibit noise and numerous features, which results in complex contour trees and inaccurate domain segmentations. To address this issue, persistent homology [24] is used to assign a significance measure, called the persistence, to each edge of a contour tree (Sec. 2.3.6). The persistence of an edge can then be used to recursively collapse edges—which represent feature groups—onto more significant edges they are attached to—which represent the parents of these feature groups (Fig. 2.6c). This process can be used to simplify the segmentation or even the original scalar field, which enables a robust hierarchical feature characterization (Fig. 2.6d).

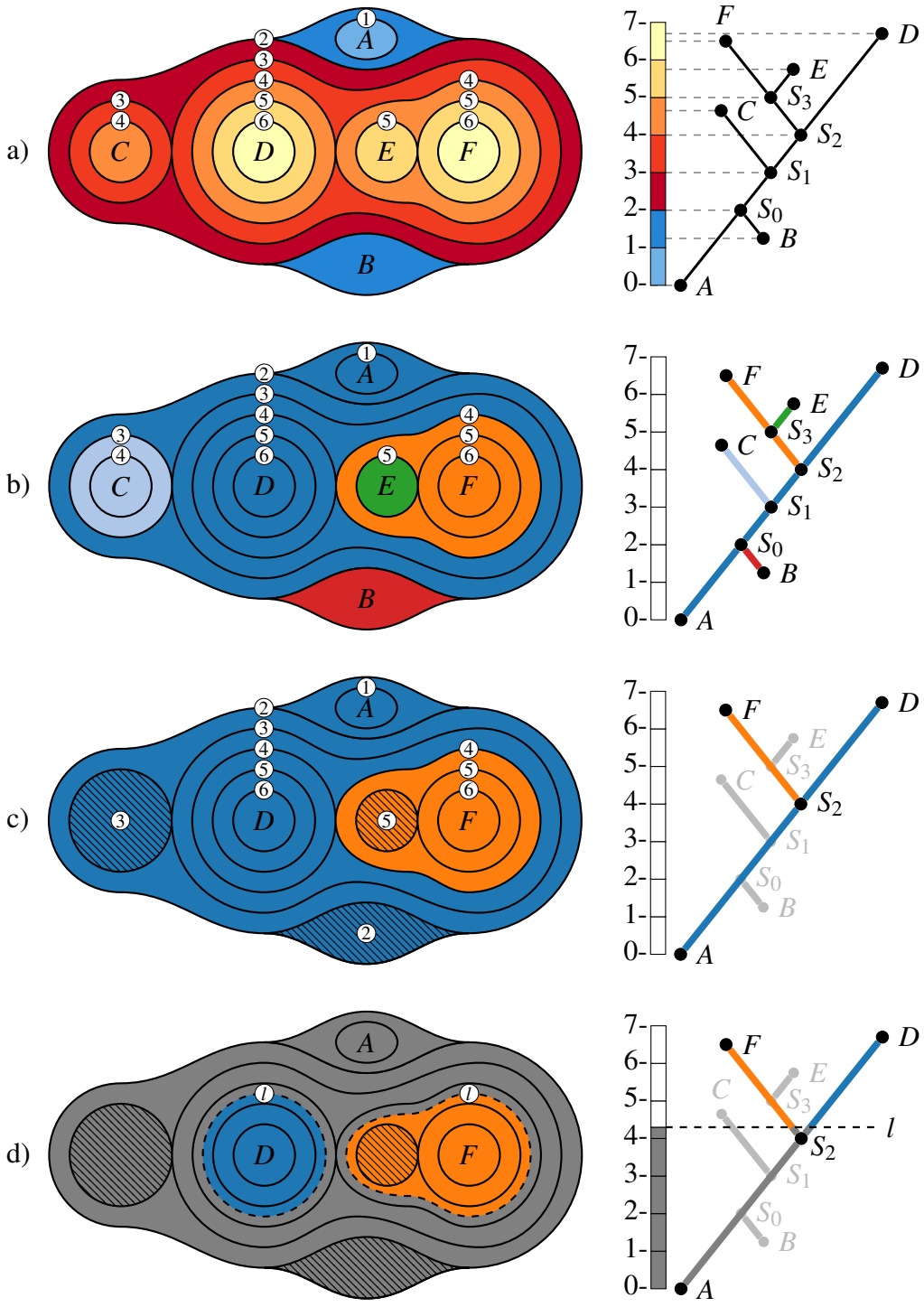


Figure 2.6: Overview of the contour tree segmentation-based feature characterization: based on the contour tree of a scalar field (a), the proposed method derives a branch decomposition (b), which is filtered dynamically based on persistence (c) to derive significant sublevel, level, and superlevel sets (d).

2.3.1 Level, Sublevel, and Superlevel Sets

In many applications, interesting features in scalar fields $f : \mathbb{X} \to \mathbb{R}$ can be characterized as domain subsets for which the assigned values are smaller, equal, or larger than a predefined value l (also called level or isovalue), and are referred to as sublevel sets $\mathcal{L}_f^-(l)$, level sets $\mathcal{L}_f(l)$, and superlevel sets $\mathcal{L}_f^+(l)$, respectively (Fig. 2.7b-d). For instance, superlevel sets correspond to highly turbulent regions in flow fields (vortices), areas in combustion simulations above a fuel consumption rate threshold (burning regions), parts of the universe exceeding a certain dark matter density (halos), regions of a fluid with a minimum viscosity (viscous fingers), or city blocks exhibiting elevated crime levels (crime hotspots). The same holds true for areas with the exact level value (level sets), and areas below a threshold (sublevel sets). To distinguish individual features—e.g., to differentiate between particular crime hotspots or dark matter halos—a component segmentation $\dot{\mathcal{S}}$ assigns to each connected component of the sets a unique label (colored regions of Fig. 2.7). Connected components of level sets—also called contours—are of special interest as they represent feature boundaries, i.e., the boundaries of sublevel and superlevel set components. Depending on which set is actually used, a segmentation is respectively called a sublevel or superlevel set component segmentation.

Definition 38 (Level / Sublevel / Superlevel Set) *For a scalar field $f : \mathbb{X} \to \mathbb{R}$ and a level $l \in \mathbb{R}$ (also called isovalue), the level, sublevel, and superlevel sets are defined as*

$$\mathcal{L}_f(l) = \{x \in \mathbb{X} \mid f(x) = l\}$$
$$\mathcal{L}_f^-(l) = \{x \in \mathbb{X} \mid f(x) \leq l\}, \text{ and}$$
$$\mathcal{L}_f^+(l) = \{x \in \mathbb{X} \mid f(x) \geq l\}, \text{ respectively.}$$

Definition 39 (Contour) *A contour is a connected component of a level set.*

Definition 40 (Component Segmentation) *A component segmentation $\dot{\mathcal{S}} : \mathbb{X} \to \mathbb{Z}$ for an enumeration $(\dot{\mathcal{L}}_0, \ldots, \dot{\mathcal{L}}_{n-1})$ of n distinct connected components $\dot{\mathcal{L}}_i \subseteq \mathbb{X}$ is defined as*

$$\dot{\mathcal{S}}(x) = \begin{cases} i, & \text{if } x \in \dot{\mathcal{L}}_i \text{ for some } i \\ -1, & \text{otherwise.} \end{cases}$$

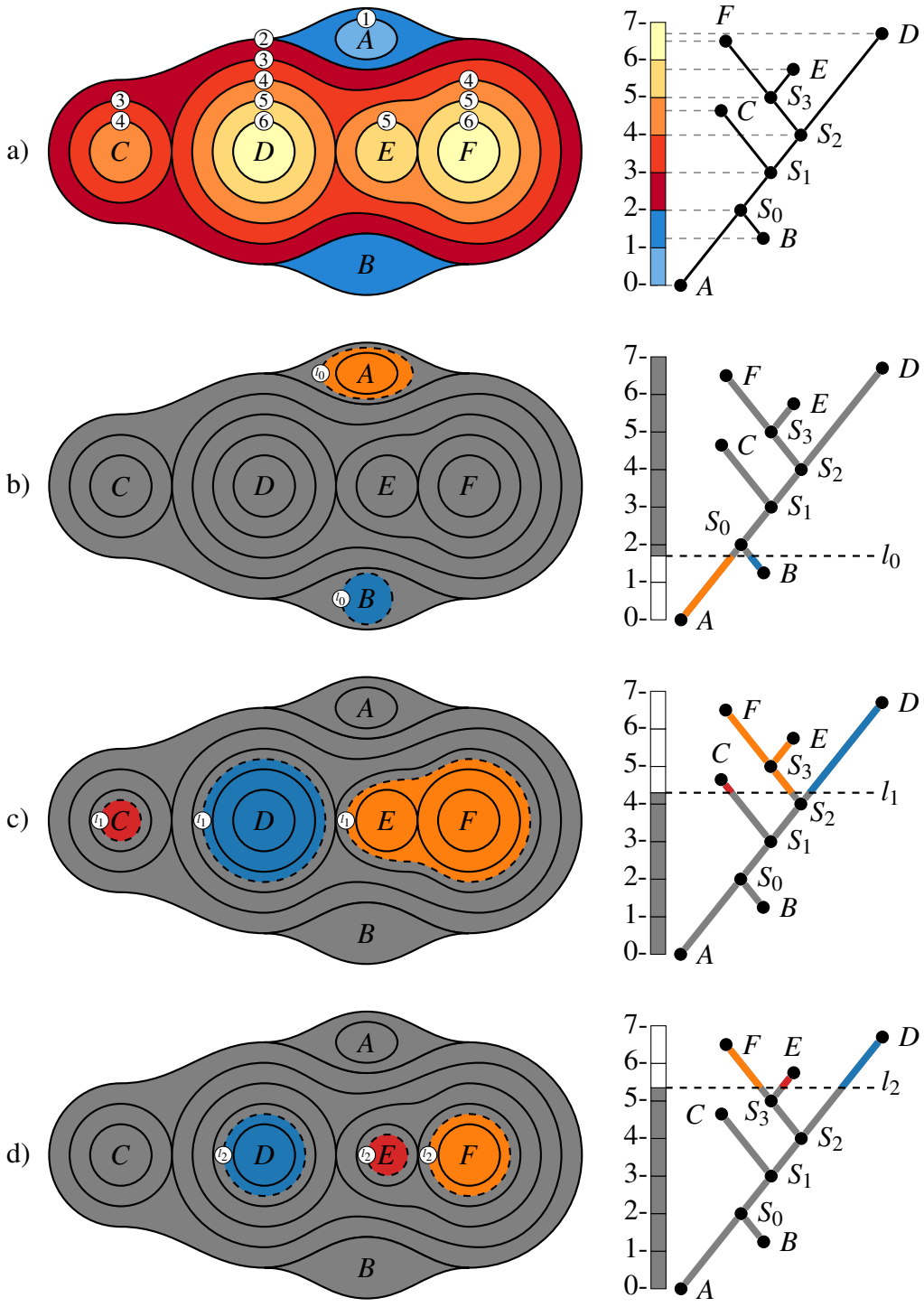


Figure 2.7: Illustrations of an example scalar field (a), a sublevel set component segmentation (b), and two superlevel set component segmentations for different levels (c-d). The segmentations show that the chosen level has a significant impact on the number and shape of features. Note, contours for these examples correspond only to the dashed lines without their enclosed regions. The evolutions of these contours during a level sweep are represented by the contour tree (right), which is introduced in the rest of this section.

Computing Component Segmentations

To derive a component segmentation on a PL manifold $\mathcal{M}$ based on a scalar field f and a level l, one can iterate over the vertices of $\mathcal{M}$ in two phases. First, all vertices that are inside a component are assigned an intermediate label, and all other vertices are labeled as the background. This is done very efficiently by probing f at the vertex locations. The next iteration assigns a unique label to each edge connected group of vertices that have an intermediate label. Note, this process only labels the vertices that are contained in the components without computing the actual boundary geometry of the components (which is describe in the next section). However, this simple algorithms is often sufficient.

Specifically, the procedure *ComputeCS(f, $\mathcal{M}$, l, n, m, $\mathcal{V}$)* outlined in Alg. 1 derives either a sublevel or superlevel set component segmentation depending on whether the mode m is set to 1 or -1, respectively. To this end, it detects components, labels them with a unique integer starting at n, and also represents each component via a vertex that is inserted into a separate vertex set $\mathcal{V}$ that holds these component representatives. Specifically, the lines 2-4 assign to each vertex either the value -1 if they belong to the background, or the intermediate label -2 if they are contained in some component. Subsequently, the lines 5-9 iterate again over all vertices and search for the intermediate label -2, which indicates an unlabeled component. For each such unlabeled component, the subprocedure *FloodFill(S, $\mathcal{M}$, v, n)* assigns the current label n to any vertex that is connected to the seed vertex v through a path on the edges of $\mathcal{M}$ that only includes vertices with the labels -2 or n. This subprocedure also returns a new vertex located at the center of mass of the labeled vertices that represents the corresponding component. Next, this vertex is inserted into $\mathcal{V}$, and n is increased by one to uniquely label the next component. Finally, the complete domain segmentation is returned in line 10.

Algorithm 1: ComputeCS(PLSF f, PLM $\mathcal{M}$, Level l, Label n, Mode m, Vertices $\mathcal{V}$)

1 $S \leftarrow [\,]$ *// Initialize Component Segmentation*
2 *// Compute Sublevel or Superlevel Sets based on m*
3 **foreach** vertex $v \in \mathcal{M}$ **do**
4 $\quad S[v] \leftarrow \begin{cases} -2 & \text{if } (m > 0 \wedge f(v) \leq l) \vee (m < 0 \wedge f(v) \geq l) \\ -1 & \text{otherwise} \end{cases}$
5 *// Label Individual Components*
6 **foreach** vertex $v \in \mathcal{M}$ **do**
7 $\quad$ **if** $S[v] = -2$ **then**
8 $\quad\quad \mathcal{V} \leftarrow \mathcal{V} \cup \{$ FloodFill(S, $\mathcal{M}$, v, n) $\}$
9 $\quad\quad n \leftarrow n+1$
10 **return** S

Triangulating Level Sets on Piecewise Linear Manifolds

There are several algorithms that derive sublevel, level, and superlevel set triangulations of PL scalar fields defined on PL d-manifolds for $d \leq 3$. A predominant example for three-dimensional PL manifolds is the *Marching Tetrahedra* algorithm [22], which is a variation of the original *Marching Cubes* algorithm [59] that uses tetrahedra (3-simplices) instead of cubes to represent three-dimensional cells. Their counterparts for two-dimensional manifolds are referred to as *Meandering Triangles* and *Marching Squares*, accordingly. Even more recent isocontouring algorithms [17, 93, 109] are still based on the same principle, which is summarized in the following.

The core concept of the marching tetrahedra algorithm is to independently check for each tetrahedron if a level set passes through it by determining which vertices have smaller or larger values than the corresponding level. This check results in 16 possible scenarios, which can be reduced by summarizing symmetric vertex configurations to 3 representative cases: a) no vertex value is smaller, b) one vertex value is smaller, and c) two vertex values are smaller than a specified level (Fig. 2.8). In the first case, a tetrahedron does not contribute any part of the level set surface. For the second and third case, a tetrahedron is partitioned into two parts by a triangulation (blue triangles of Fig. 2.8) that separates the smaller from the larger vertices. This procedure can be efficiently implemented via a lookup table that returns for a vertex configuration the corresponding triangulation. Then, to determine the actual geometry of the level set, the vertex positions of the triangulation (white vertices of Fig. 2.8) are interpolated on the edges of the tetrahedron based on the associated PL scalar field (colored vertices of Fig. 2.8). Finally, the collection of all triangulations yields a piecewise linear approximation of the complete level set.

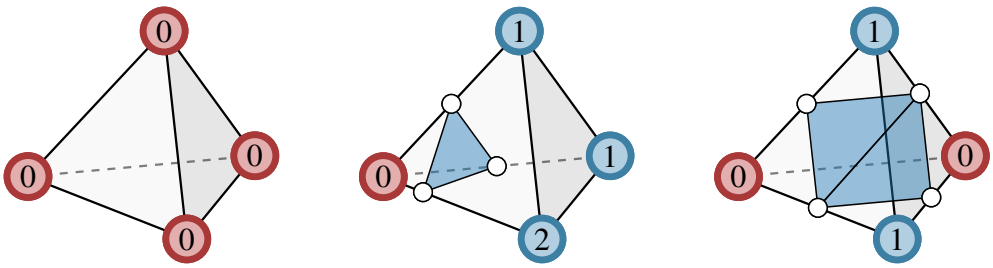


Figure 2.8: The three distinct types of vertex configurations of the *Marching Tetrahedra* algorithm for a level set with level 0.5.

2.3.2 Merge and Contour Trees

Although sublevel, level, and superlevel sets characterize significant regions in scalar fields, their number and geometry depend heavily on the chosen level (Fig. 2.7c-d). To understand this dependency, it is necessary to identify levels for which individual contours appear, disappear, merge, split, or simply vary in shape. Tracking the evolution of contours is the fundamental principle that enables the partition of the domain into homogeneous regions that do not change topologically for certain level intervals, and to aggregate these regions in a topological abstraction, called the contour tree [108].

Concept

In brief, the contour tree records the evolution of contours during a monotone level sweep. The sweep process can be thought of as a landscape where water is continuously rising from the lowest to the highest point (Fig. 2.9a left). Thus, connected parts of the landscape that are below, at, or above the current water level correspond to sublevel, level, and superlevel set components, respectively. As the water level rises, the connectivity (topology) of the components changes when the water level reaches so-called critical points, i.e., minima, maxima, and saddles for which components appear, disappear, and merge/split, respectively. For instance, consider the evolution of sublevel set components in Fig. 2.9b. First, two separate components appear at the minima A and B, which then become connected exactly when the water level passes the critical value 2. If components merge, then the resulting component inherits the label of the oldest component—in this case A—which is referred to as the elder rule [25] (encoded by color in Fig. 2.9b-d). The resulting component grows until the entire landscape is under water, which occurs when the level passes the global maximum D. The evolution of these sublevel set components—i.e., when they appear and join—is represented by the so-called *join tree* (Fig. 2.9b right). Conversely, the *split tree* records the evolution of superlevel set components, i.e., when areas above the water level split and disappear (Fig. 2.9c). In this example, the complete landscape is at first above the water level, which then sequentially splits at saddles into islands that disappear as soon as the water level passes their corresponding maxima. This can also be interpreted as a reverse sweep that tracks the merging of superlevel set components. Therefore, join and split trees are also referred to as *merge trees*, as they both describe the merge of components; just for opposite sweep directions. The *contour tree*—which summarizes the join and split tree—can be derived similarly by observing the connectivity of contours (Fig. 2.9d).

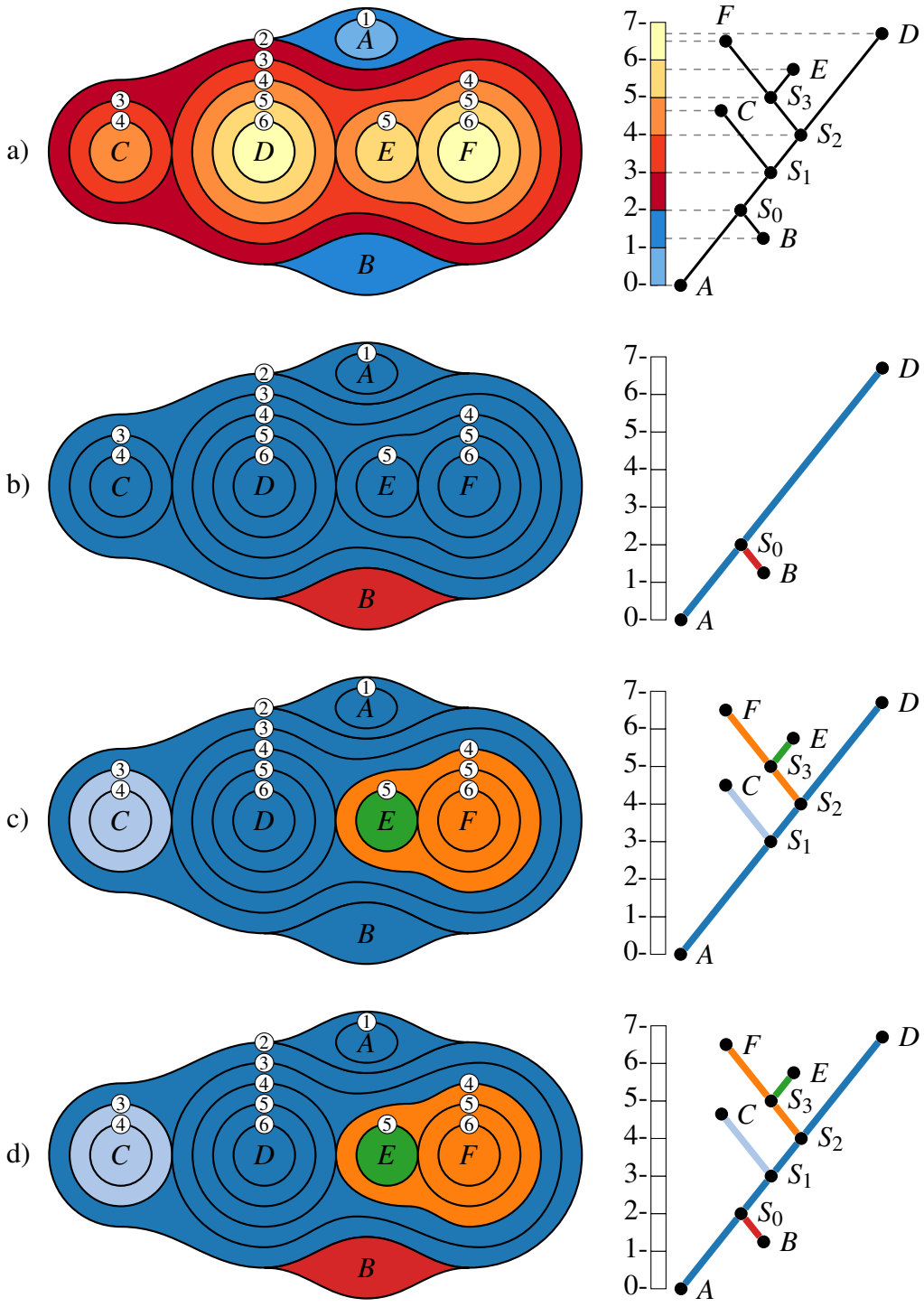


Figure 2.9: The join, split, and contour tree (b-d right), as well as their respective domain segmentations (b-d left) for an example scalar field (a). Sublevel and superlevel set components are labeled by their corresponding minima and maxima, respectively. If components merge or split, they pass on labels based on their lifetime, i.e., a merged sublevel set component inherits the label of the oldest merging component, and a splitting superlevel set component passes its label on to the component that will disappear last.

Formal Description

Now that contour trees have been introduced conceptually, the rest of this section formally describes the topological evolution of contours of scalar fields defined on manifolds. If a manifold is not required to be simply connected—i.e., if it exhibits holes, voids, and so forth—then the study of topological changes of contours translates to the more general notion of Reeb graphs [87]. Similar to contour trees, Reeb graphs represent the evolution of contours via edges and vertices, but they are also allowed to have cycles (loops). It has been shown that Reeb graphs for simply connected manifolds are loop free [18], in which case they are referred to as contour trees.

Formally, the Reeb graph $\mathcal{R}_f$ represents the evolution of contours during a continuous value sweep of a scalar field f defined on a manifold $\mathbb{M}$. The definition of the Reeb graph is based on an equivalence relation $\sim$ that assigns two points of $\mathbb{M}$ to the same equivalence class iff they have the same value, and are elements of the same contour. Imagine that these classes across all possible levels get individually contracted to a single point, i.e., individual contours (equivalence classes) are represented via distinct points that constitute curves along the increasing function values (Fig. 2.10). The graph that results from this contraction is the Reeb graph $\mathcal{R}_f$, which is formalized via the quotient space of $\mathbb{M}$ under the contour equivalence relation $\sim$ (Eq. 2.2). If $\mathbb{M}$ is simply connected, then the Reeb graph has no cycles; in which case the graph is called a contour tree $\mathcal{C}_f$. Symmetrically, the join tree $\mathcal{C}_f^-$ and the split tree $\mathcal{C}_f^+$ (collectively referred to as merge trees) are defined in the same manner for equivalence relations based on sublevel and superlevel set components, respectively. A common representation of these graphs are one-dimensional simplicial complexes that triangulate their corresponding quotient space.

A split, join, or contour tree $\mathcal{C}_f^*$ is associated with a domain segmentation $\phi : \mathbb{M} \to \mathcal{C}_f^*$ that maps any point of $\mathbb{M}$ to its corresponding point in $\mathcal{C}_f^*$, and the tree scalar field $\psi : \mathcal{C}_f^* \to \mathbb{R}$ that maps any equivalence class of $\mathcal{C}_f^*$ to the corresponding function value. Moreover, the trees can be partitioned into groups of connected edges to derive a so-called branch decomposition (Def. 43). Hence, a so-called merge/contour tree segmentation $\tilde{\mathcal{S}} = (\mathcal{C}_f^*, \phi, \psi)$ completely partitions the domain into connected regions that correspond to branches of $\mathcal{C}_f^*$ (Fig. 2.9; Def. 44). These abstractions are essential to characterize individual features based on level intervals, and to project the graphs onto new topological spaces, e.g., to derive a spatial embedding (Fig. 2.10), or an optimized layout (Fig. 2.9, right).

The methodology described in this manuscript focuses on merge and contour trees. An overview of additional theory and algorithms for Reeb graphs can be found in the book by Edelsbrunner and Harer [25], and the habilitation thesis of Tierny [103].

Definition 41 (Quotient Space) *Let $\sim$ be an equivalence relation on a topological space $\mathbb{X}$ with topology T_X; let Y be the set of all equivalence classes of $\sim$ for $\mathbb{X}$; and let $\phi : \mathbb{X} \to Y$ be a surjective function that maps each element $x \in \mathbb{X}$ to its equivalence class $[x] \in Y$. Then, the topological space $\mathbb{Y}$ for set Y and the quotient topology*

$$T_Y = \{\, y \subseteq Y \mid \phi^{-1}(y) \in T_X \,\} \tag{2.1}$$

is called the quotient space, which is denoted by $\mathbb{Y} = \mathbb{X}/\sim$. Hence, the quotient topology T_Y is the set of subsets $y \subseteq Y$ whose preimages $\phi^{-1}(y)$ are open sets of $\mathbb{X}$.

Definition 42 (Reeb Graph; Contour, Merge, Join, and Split Tree) *For a scalar field f defined on a manifold $\mathbb{M}$, let $\dot{\mathcal{L}}_{f,x}(i)$ denote the contour of level set $\mathcal{L}_f(i)$ that contains the point $x \in \mathbb{M}$, and let $\sim$ be an equivalence relation between two points $u, v \in \mathbb{M}$ such that*

$$u \sim v \iff [\, f(u) = f(v) \;\wedge\; u \in \dot{\mathcal{L}}_{f,v}(\, f(v)\,)\,]. \tag{2.2}$$

*Then, the **Reeb graph** $\mathcal{R}_f$ is defined as the quotient space $\mathbb{M}/\sim$. Symmetrically, the **join tree** $\mathcal{C}_f^-$ and the **split tree** $\mathcal{C}_f^+$ are defined in the same manner for equivalence relations based on sublevel and superlevel set components, respectively. Join and split trees are also referred to as **merge trees**. If $\mathbb{M}$ is simply connected, then the Reeb graph is loop free and is called a **contour tree** $\mathcal{C}_f$.*

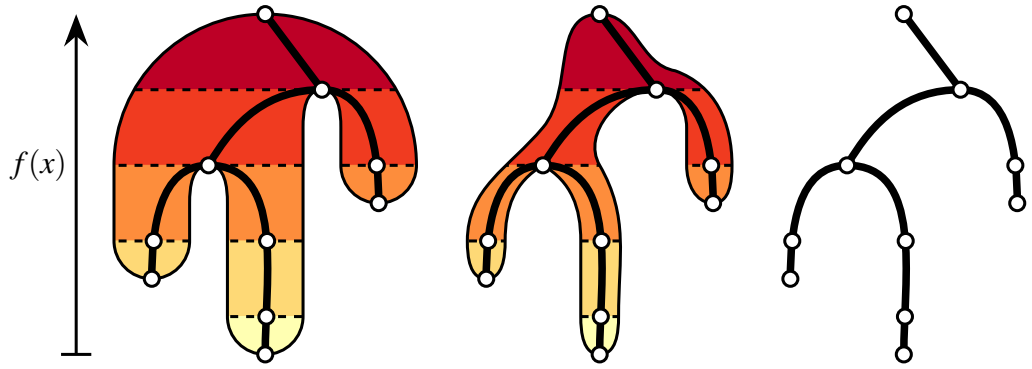


Figure 2.10: Illustration of the contraction process (left to right) of a 2-manifold $\mathbb{M}$ with an associated height function (left) to the corresponding contour tree (right). This process is formalized via the quotient space $\mathbb{M}/\sim$ for a contour equivalence relation $\sim$ (Eq. 2.2). Some example equivalence classes of $\sim$ are shown by dashed lines, where each class gets contracted to a single point (white discs). The union of all contracted points constitute the underlying space of a graph; in this case the contour tree.

Definition 43 (Branch Decomposition) *A branch decomposition of a one-dimensional simplicial complex $\mathcal{K}$ is an enumeration $\mathcal{B}$ of connected simplicial complexes of $\mathcal{K}$ s.t.*

(i) the union of all branches $\bigcup \mathcal{B}$ is equal to $\mathcal{K}$; and

(ii) the intersection of any two distinct branches of $\mathcal{B}$ is either empty or a single vertex.

Definition 44 (Merge / Contour Tree Segmentation) *A merge/contour tree segmentation* $\tilde{\mathcal{S}} = (\mathcal{C}_f^*, \phi, \psi)$ *of a scalar field* f *defined on a simply connected PL manifold* $\mathcal{M}$ *consists of a merge or contour tree* $\mathcal{C}_f^*$*, the corresponding domain segmentation* $\phi : \mathcal{M} \to \mathcal{C}_f^*$ *that maps any point of* $\mathcal{M}$ *to its associated point in* $\mathcal{C}_f^*$*, and the tree scalar field* $\psi : \mathcal{C}_f^* \to \mathbb{R}$ *that assigns to any point of* $\mathcal{C}_f^*$ *the corresponding value of* f*.*

2.3.3 Critical Points

Critical points are essential for computing merge and contour trees of scalar fields. For a smooth function, critical points are located at points for which the gradient of the function vanishes.* However, PL scalar fields require an alternative definition since their gradient is piecewise constant. Such an alternative definition can be inferred from Morse theory [72], which was originally derived for a special type of smooth real-valued functions—called Morse functions—that are required to have non-degenerate† critical points with distinct values.

The key result of Morse theory is that the topology of sublevel sets only change in the local neighborhood of critical points [5, 72], i.e., when a level passes a critical value (as demonstrated with the example of Fig. 2.9). This fact is the basis for the definition of critical points of a PL scalar field f given on a PL manifold $\mathcal{M}$. First, note that if it is required that critical points of f must have distinct values, then critical points can only be located at vertices of $\mathcal{M}$. This follows form the fact that any point inside a higher dimensional simplex is mapped to a constant gradient vector.‡ A vertex $v \in \mathcal{M}$ with function value $l = f(v)$ can then be classified solely based on the topology of its local neighborhood. Specifically, the connected components of its lower and upper link correspond to parts of the sublevel set $\mathcal{L}_f^-(l-\varepsilon)$ and superlevel set $\mathcal{L}_f^+(l+\varepsilon)$ for some small $\varepsilon > 0$, respectively. If the lower or upper link is empty, then v must be a local minimum or maximum, respectively (Fig. 2.11a-b). Otherwise, if the upper and lower link are both single components, then the new sublevel and superlevel set components for level l simply grow by absorbing v without a change in connectivity, and v is therefore called a regular point (Fig. 2.11c). Finally, if the lower link consists of multiple components, then the new sublevel set for level l will connect them locally at v, which corresponds to a change in topology at a saddle point (Fig. 2.11d). This applies symmetrically for multiple

*The smooth setting requires numerous additional definitions; such as charts, smooth functions, and differentiable manifolds. As the smooth case is not in the scope of this manuscript, it suffices to say that in this setting it is possible to compute the continuously differentiable gradient of a function, which corresponds to a vector pointing locally to the largest increase in function value.

†Degenerate critical points are defined later in the text.

‡The special case where the gradient is the zero vector implies that all points inside a simplex have the same value, and thus can not be critical points by definition as they are required to have distinct values.

upper link components and superlevel sets. Additionally, if the lower or upper link has more than two components, then v is called a degenerate multi-saddle (Fig. 2.11e). Note, some saddles might locally connect parts of the same component, and thus not change the connectivity of the level set (Fig. 2.11f). These degenerate saddles are called *false* saddles and are treated as regular vertices during merge tree computation [18, 36, 105].

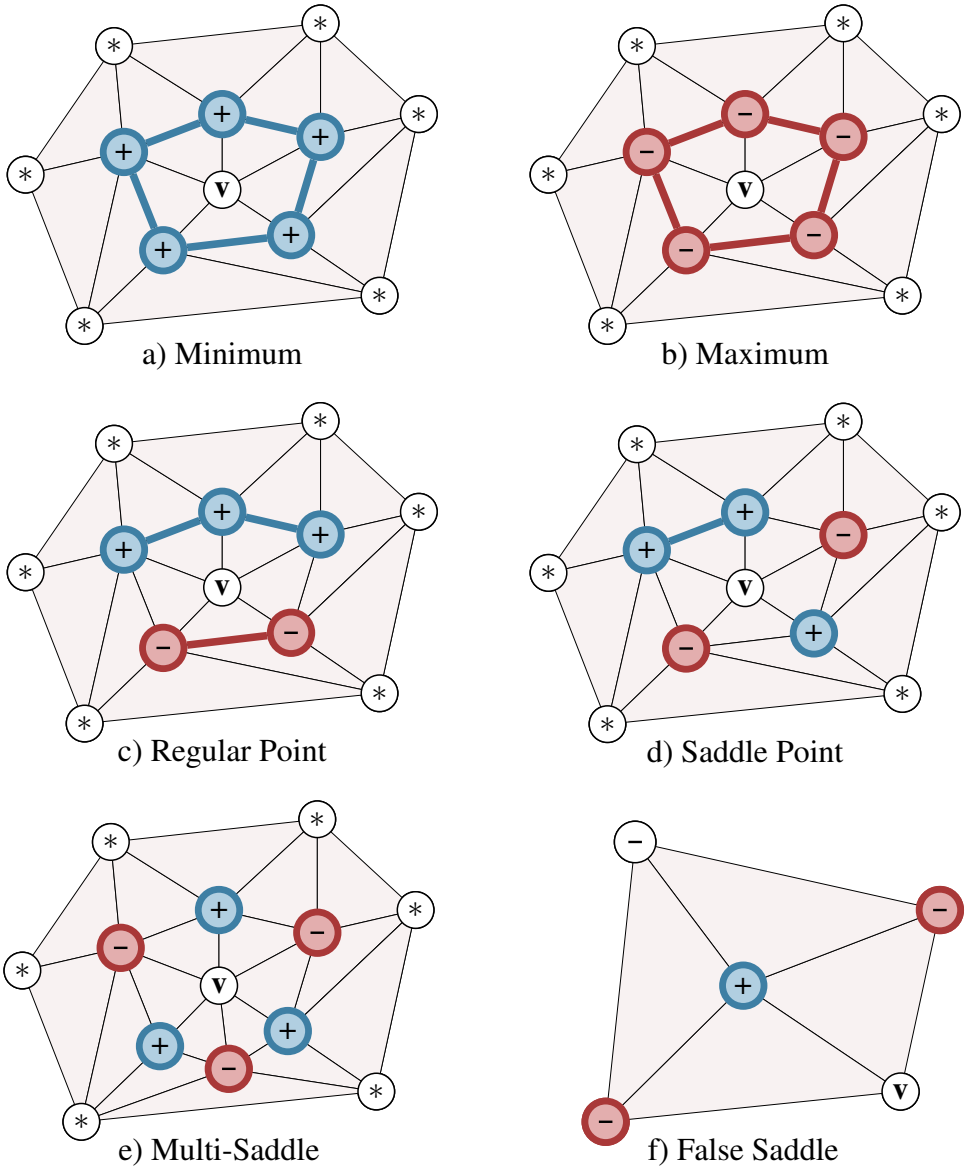


Figure 2.11: Classification of a vertex v inside a piecewise linear Morse scalar field based on the connectivity of its lower link (red), and upper link (blue). If the lower or upper link is empty, then v is a minimum (a) or maximum (b), respectively. If both links are single connected components, then v is a regular point (c). Otherwise at least one link consists of more than one component, in which case v is a saddle (d-e). If a saddle would locally connect only the same globally connected component, then the saddle is called a false saddle (f). Note, this can not be determined locally.

Definition 45 (Regular Point) *A regular point of a PL scalar field is a vertex whose upper and lower link are both non empty and simply connected.*

Definition 46 (Minimum / Maximum) *A minimum or maximum of a PL scalar field is a vertex with an empty lower or upper link, respectively.*

Definition 47 (Saddle Point) *A k-saddle of a PL scalar field is a vertex v for which either its lower or upper link consists of $k+1$ components, in which case v is called a join or split saddle, respectively. Saddles with $k > 1$ are referred to as multi-saddles or degenerate critical points.*

However, to substantiate the previous classification of points—i.e., to transfer the results of Morse theory—it is necessary to define the equivalent of a Morse function in the PL setting. Hence, a PL scalar field is called a PL Morse scalar field iff (i) all its critical values are distinct, and (ii) all critical points are non degenerate. These constraints respectively ensure that sublevel set components appear, merge, and disappear only at vertices with distinct values, and that at most two components merge for a critical value. Obviously, such scalar fields rarely occur in practice, but it is possible to modify a PL scalar field f given on a PL manifold $\mathcal{M}$ such that it fulfills these requirements. The first condition is satisfied if the restriction of f to the vertices of $\mathcal{M}$ is injective. This can be enforced by using simulation of simplicity (SoS) [31] to break ties between two vertices by comparing their respective coordinates or index in an enumeration. The second condition is violated if a saddle point is a multi-saddle, e.g., a monkey saddle which connects three sublevel sets. In this case, the star of a multi-saddle can be re-triangulated into multiple regular saddles via a method called multi-saddle unfolding [25]. Thus, it is reasonable to assume that any piecewise linear scalar field can be modified to fulfill the criteria of a piecewise linear Morse scalar field.

Definition 48 (Piecewise Linear Morse Scalar Field) *A piecewise linear scalar field is called a piecewise linear Morse scalar field iff*

(i) all critical points have distinct values, and;

(ii) all critical points are non degenerate.

2.3.4 Merge Tree Computation

This section defines an algorithm for computing the merge tree of a PL Morse scalar field f defined on a PL manifold $\mathcal{M}$. Even modern algorithms that utilize parallelism [14, 36, 52, 69, 73, 74] are still based on the principle that the topology of sublevel sets only changes at critical points (Sec. 2.3.3). Thus, one way of computing the join tree is to first sort all vertices by value in ascending order, and then iterate over all vertices to sequentially add them to groups that represent the growing sublevel set components. These groups represent the evolution of sublevel set components during the iteration, i.e., they appear at minima, grow for regular vertices, and merge at saddles that connect two before disconnected components. This can be efficiently implemented via a union-find data structure that maintains these groups [13]. The split tree of f corresponds to the join tree of the inverse field $-f$, and can therefore be computed with the same procedure.

Alg. 2 outlines a procedure that simultaneously derives a branch decomposition $\mathcal{B}$ and a spatial embedding of the join tree C_f^-. Specifically, the algorithm initializes $\mathcal{B}$ as an empty set, creates a new union-find data structure $\mathcal{G}$, sorts all vertices of $\mathcal{M}$ in ascending order, and then iterates over all vertices of the resulting vertex enumeration $\bar{V}$ to construct groups and branches. The vertices with the lowest and highest scalar value of a group $G \in \mathcal{G}$ are respectively denoted by $\alpha(G)$ and $\omega(G)$. At the beginning of each iteration it is assumed that

(i) every sublevel set component of the previous iteration is represented by a unique group of $\mathcal{G}$, and vice versa; and

(ii) for each group $G \in \mathcal{G}$ exists exactly one branch $B \in \mathcal{B}$ that contains $\alpha(G)$ and $\omega(G)$.

Each iteration processes a vertex $v \in \bar{V}$ based on its type, which can be determined by the number of its lower link components C (Definitions 45–47). Specifically, if C is empty, then v is a minimum. Thus, a new group and a new branch are created that both contain v (line 6-8). Otherwise, C contains one or two lower link components, where each component is a subset of exactly one (possibly even the same) sublevel set component of the previous iteration. Based on iteration assumption (i), line 11 retrieves for each lower link component $c \in C$ the unique group $G \in \mathcal{G}$ that contains c. Next, the subprocedure *GetBranch* returns the unique branch $B \in \mathcal{B}$ that contains $\alpha(G)$ and $\omega(G)$, which must exist due to iteration assumption (ii). Line 13 then extends this branch by v and the edge $\langle \omega(G), v \rangle$ that connects v to the previously inserted vertex of G. Always choosing and extending the oldest branch is known as employing the elder rule [25]. Then, v is also inserted into G. Finally, line 15 actually checks if C contains two components that belong to different sublevel set components of the previous iteration. If this is the case, it is necessary to unify their respective groups. If the two components belong to

the same sublevel set component, then v is called a false saddle as the connectivity of the new component does not change by absorbing v, and therefore no further action is necessary. Since each iteration preserves the iteration assumption, the algorithm can proceed by induction until all vertices are processed, which yields the complete branch decomposition $\mathcal{B}$. A formal proof that the resulting graph $\mathcal{C}_f^+ = \bigcup \mathcal{B}$ is indeed the join tree can be found in the work of Carr et al. [13]. This procedure can also be used to derive the split tree by processing the inverted scalar field, or by symmetrically tracking the connectivity of superlevel set components and upper link components.

The branches $\mathcal{B}$ yield a complete merge tree segmentation $\tilde{\mathcal{S}} = (\mathcal{C}_f^*, \phi, \psi)$ (Def. 44), where the merge tree $\mathcal{C}_f^* = \bigcup \mathcal{B}$ is the union of all branches, and the domain segmentation ϕ and tree scalar field ψ are defined explicitly by $\mathcal{C}_f^*$ since the tree contains all vertices of $\mathcal{M}$ (Fig. 2.12). In this case, $\mathcal{C}_f^*$ is called an augmented merge tree. A task-parallel version of this algorithm [36] is implemented in the *Topology ToolKit* [104]. In the following chapters, merge tree segmentations are used to iteratively simplify $f = \psi \circ \phi$ to make feature identification more stable, and to implement linking and brushing between the trees and the feature domain.

Algorithm 2: ComputeAugmentedJoinTree(PLMSF f, PLM $\mathcal{M}$)

```
1  B ← ∅   // Initialize Branch Decomposition
2  G ← NewUnionFind()
3  V̄ ← SortVertices( M, f )
4  foreach vertex v ∈ V̄ do
5      C ← GetLowerLinkComponents( v, M, f )
6      if Size(C) = 0 then
7          NewGroup( G, v )
8          B ← B ∪ {v}
9      else
10         foreach component c ∈ C do
11             G ← FindGroup( G, c )
12             B ← GetBranch( B, α(G) )
13             B ← B ∪ {⟨ω(G), v⟩, v}
14             G ← G ∪ v
15         if Size(C) = 2 ∧ GetGroup( G, C[0] ) ≠ GetGroup( G, C[1] ) then
16             UnifyGroups( G, C )
17 return B
```

The middle of Fig. 2.12 illustrates the join tree computation for the running example. First, a blue and a red branch appear for the minima $(A : 0)$ and $(B : 1.2)$, respectively. Then, the vertices with value 2 are sequentially added to the first branch where ties in the sorting order are broken by simulation of simplicity (SoS) [31]. Thus, one of these vertices (S_0) will connect the two branches at some point during the iteration. The concrete SoS implementation determines which vertex is actually chosen. Here, no other vertex is added to branch B until it merges with branch A. The remainder of the algorithm simply adds the remaining vertices and corresponding edges to the oldest branch they are connected to, in this case A. The split tree is computed symmetrically by executing Alg. 2 with the inverted scalar field (Fig. 2.12 bottom). Hence, four distinct branches appear for the maxima $(D : 6.7)$, $(F : 6.5)$, $(E : 5.7)$, and $(C : 4.7)$, which sequentially merge at the saddle points $(S_3 : 5)$, $(S_2 : 4)$, and $(S_1 : 3)$. The oldest branch then absorbs the remaining vertices in descending order, here D.

The described procedure automatically derives a spatial embedding of the merge tree (Fig. 2.12 left). It is also possible to compute an optimized tree layout to minimize the number of edge crossings and arrange edges based on the corresponding branch decomposition [84] (Fig. 2.12 right). Moreover, vertices of merge trees can again be reclassified based on the number of edges to vertices with lower value (called down arcs), and higher value (called up arcs). Thus, vertices with no down or up arc in at least one of the trees correspond to minima and maxima, respectively. These vertices have a special context for a specific tree type, i.e., minima are leafs in the join tree, and maxima are leafs in the split tree. Additionally, the global maximum and minimum are the respective roots of join and split trees. A vertex is a regular point iff it has exactly one down and one up arc in both trees. Vertices with more than one down or up arc in at least one of the trees are called merge or split saddles, accordingly. This reclassification removes some degenerate cases, such as false saddles. Next, a common post-processing step is to remove regular points from the trees while preserving the graph connectivity in order to reduce their size. In practice, however, edges often maintain lists of their corresponding regular points to explicitly store the inverse map ϕ^{-1}. After this procedure, the split tree still contains the critical vertices of the join tree, and vice versa (nodes without labels in Fig. 2.12 right). In the next step, both merge trees can be combined to yield the contour tree.

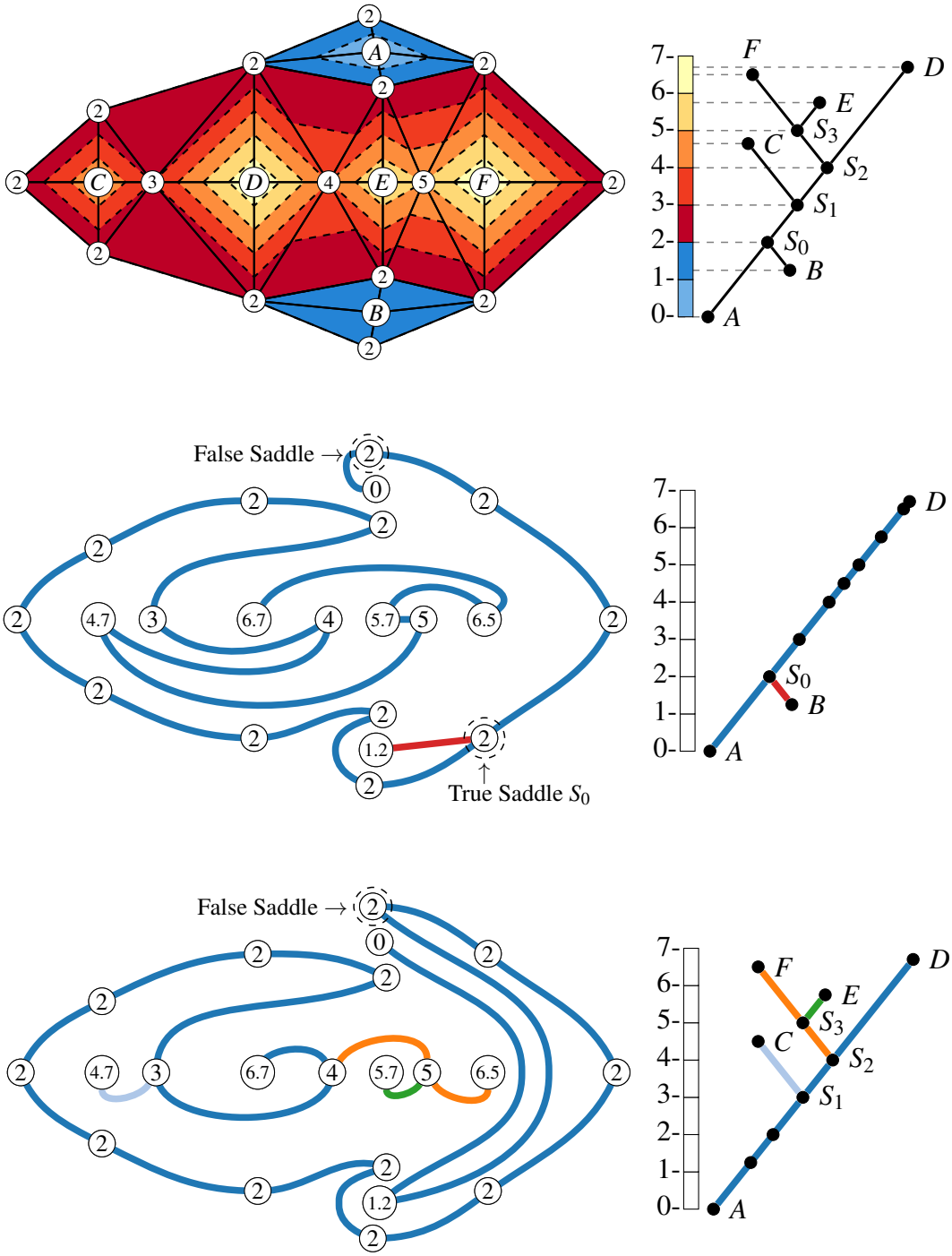


Figure 2.12: Illustration of Alg. 2 that computes the join tree (middle) and split tree (bottom) of a PL scalar field (top) by sequentially grouping/connecting vertices based on their sorted values, and connectivity. Merge tree branches (colored edges) are bent to prevent unnecessary edge crossings, and ties between vertex values in the sorting order are resolved by also considering their spatial position. In this example, an order was chosen that results in the least amount of edge crossings. Note, if two branches merge, only the oldest one is continued based on the elder rule.

2.3.5 Contour Tree Computation

Instead of explicitly computing the contour tree $\mathcal{C}_f$ based on the function f, it is possible to constructed $\mathcal{C}_f$ by iteratively adding leafs and attached edges from the merge trees to $\mathcal{C}_f$ [13]. Specifically, each iteration processes one leaf of the split or join tree that is not a split/join saddle in the other tree. Note, there always exists such a vertex as there are more leafs than join/split saddles. First, $\mathcal{C}_f$ is initialized as an empty set, and a queue is filled with the leaf vertices of the merge trees, where a superscript denotes the actual tree in which the vertex is a leaf. Lets assume a leaf v^- from the join tree $\mathcal{C}_f^-$ is next in the queue (the other case for a leaf $v^+ \in \mathcal{C}_f^+$ is symmetrical). By definition, v^- is not the root of $\mathcal{C}_f^-$ nor a split/join saddle in $\mathcal{C}_f^+$, and v^- is connected to some vertex $u \in \mathcal{C}_f^-$ via an up arc $\langle v^-, u\rangle$. Then, the simplicies v^-, u, and $\langle v^-, u\rangle$ are inserted into $\mathcal{C}_f$ while skipping duplicate simplicies. Next, v^- and $\langle v^-, u\rangle$ are removed from $\mathcal{C}_f^-$. At the same time, v^- in $\mathcal{C}_f^+$ is either connected to two vertices $x, y \in \mathcal{C}_f^+$ by a down and up arc, or v^- is connected to a single vertex $z \in \mathcal{C}_f^+$ (the current root of $\mathcal{C}_f^+$). In the first case, v^-, $\langle v^-, x\rangle$, and $\langle v^-, y\rangle$ are removed from $\mathcal{C}_f^+$, but a new edge $\langle x, y\rangle$ is inserted in $\mathcal{C}_f^+$ to preserve the graph connectivity. In the other case, v^- and $\langle v^-, z\rangle$ are simply removed from $\mathcal{C}_f^+$. This way, each iteration produces valid join and split trees with the same set of vertices, which is the required input for the next iteration. If at the end of an iteration new leafs appear that fulfill the aforementioned criteria, then they are added to the queue. The contour tree is complete when all vertices have been processed. Also note that this construction process combines the merge tree segmentations into a contour tree segmentation.[§] A proof that the constructed graph is the contour tree of f can be found in the original paper by Carr et al.[13].

Fig. 2.13 illustrates the contour tree construction process for the running example. Initially, all leafs of the join and split tree are inserted into the queue. In the first step, the leaf $\langle B^-\rangle$ and its attached edge need to be removed from the join tree, as indicated by the superscript. Hence, the simplicies $\langle B^-\rangle$, $\langle S_0\rangle$, and $\langle B^-, S_0\rangle$ from $\mathcal{C}_f^-$ are inserted into $\mathcal{C}_f$; the simplicies $\langle B^-\rangle$ and $\langle B^-, S_0\rangle$ are removed from $\mathcal{C}_f^-$; and the simplicies $\langle B^-\rangle$, $\langle A, B^-\rangle$, and $\langle B^-, S_0\rangle$ in $\mathcal{C}_f^+$ are replaced by the edge $\langle A, S_0\rangle$. As no new leafs appear, nothing is added to the queue. The next leaf $\langle F^+\rangle$ from the split tree is processed symmetrically. In step three, only the simplicies $\langle E^+\rangle$ and $\langle E^+, S_3\rangle$ are added to $\mathcal{C}_f$, as $\langle S_3\rangle$ is already an element of $\mathcal{C}_f$. This iteration also creates the new leaf $\langle S_3^+\rangle$ in $\mathcal{C}_f^+$ that is not a join saddle in $\mathcal{C}_f^-$, and is therefore added to the queue. This process continues until all vertices are processed, which yields the complete contour tree.

[§] An algorithm that can merge branch decompositions is described in the work of Pascucci et al. [84].

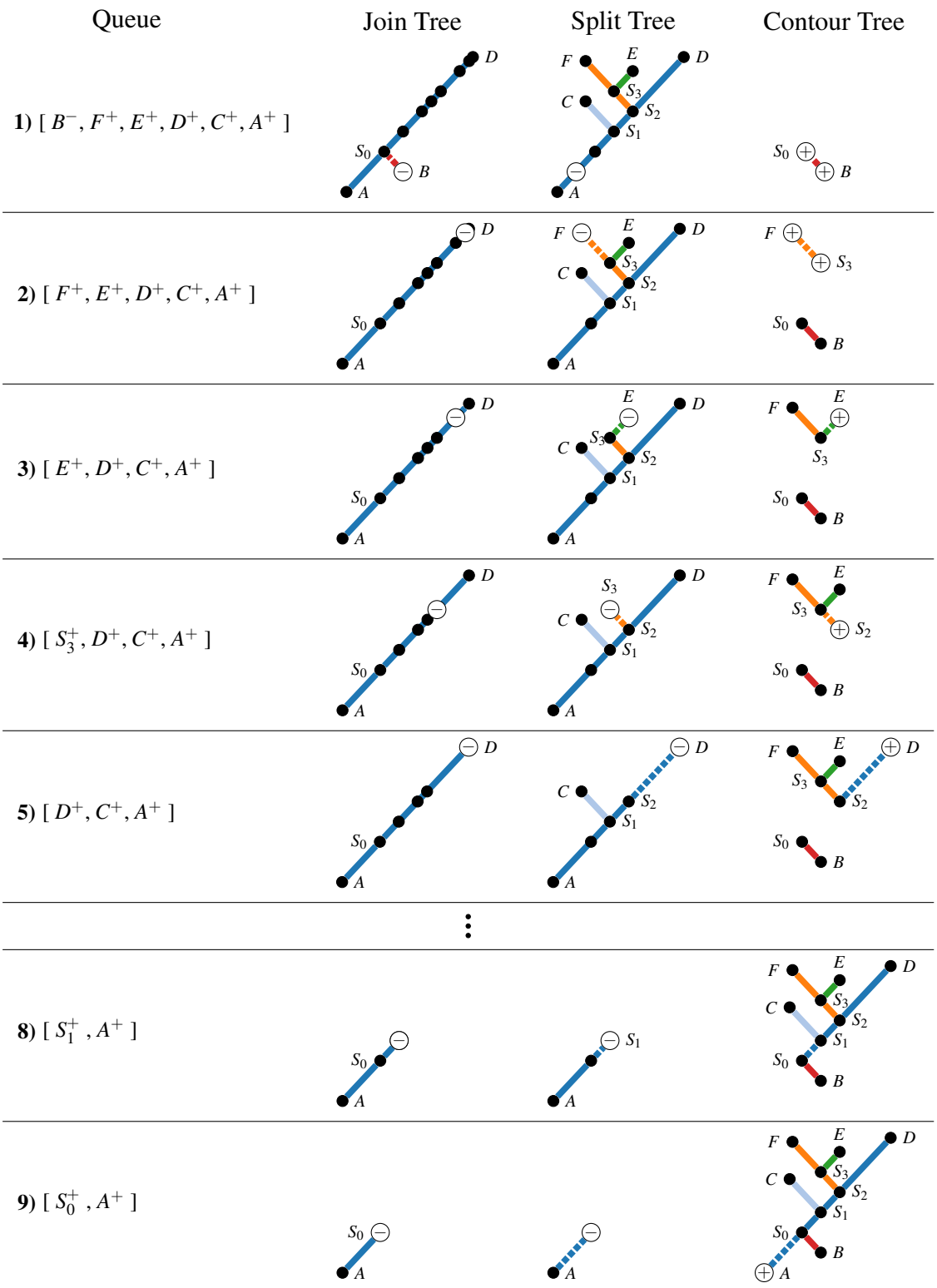


Figure 2.13: Iterative construction of the contour tree via graph operations on the join and split tree. Each iteration moves a vertex (white disc) and an attached edge (dashed line) from one merge tree to the contour tree, while updating both merge trees.

2.3.6 Topological Simplification

As demonstrated previously, merge and contour trees are excellent tools to describe the evolution of individual level, sublevel, and superlevel set components via edges between critical points. In practice, however, all components are not equally important. Especially datasets that contain noise—e.g., from numerical errors or measurement inaccuracies—exhibit slight function value perturbations that are identified as valid critical points. Therefore, the next step is to measure the significance of components (features) in order to robustly rank, select, and remove them.

Branch Persistence and Critical Point Pairs

Persistent homology [24, 29] is a topological concept that robustly measures the significance of features based on a sequence of nested sets, called a filtration. In the context of PL scalar fields over simplicial complexes, filtrations are defined as sequences of subcomplexes that correspond to sublevel and superlevel sets for monotone level sweeps. The goal of persistent homology is to measure how long individual groups exist during such sequences. Specifically, this work focuses on the lifetime (the persistence) of individual connected components. For instance, the level interval for which an individual superlevel set component first appears at a maximum, until it disappears at a minimum or saddle that connects it to an older (and therefore more persistent) component. Consider in Fig. 2.14 the maxima $(F : 6.5)$ and $(E : 5.7)$, and their corresponding superlevel set components for the level interval $(5, 6.5]$. Relatively speaking, in this interval the component containing F with lifetime 1.5 is more significant than component E that has only a lifetime of 0.7; this is also why the branch representing component E ends at the saddle, while branch F continues (Sec. 2.3.4). However, the component of F is not older (less persistent) than the component of maximum D, and so forth. This concept is symmetrically defined for the edges of join and contour trees.

Note, the presented merge and contour tree algorithms and the resulting branch decomposition $\mathcal{B}$ for a scalar field $f = \psi \circ \phi$ are already based on this concept. Thus, each branch $B \in \mathcal{B}$ is associated to a unique critical point pair (u, v)—called a persistence pair—that consists of the vertices $u, v \in B$ at the endpoints of the branch where by convention $\psi(u) \leq \psi(v)$. A common representation of these persistence pairs is a so-called persistence diagram $\mathcal{P}_\psi$, i.e., a one-dimensional simplicial complex where each persistence pair (u, v) is represented by an edge $\langle\, (\psi(u), \psi(u)) \,,\, (\psi(u), \psi(v)) \,\rangle$ and its faces (Fig. 2.14 bottom left).

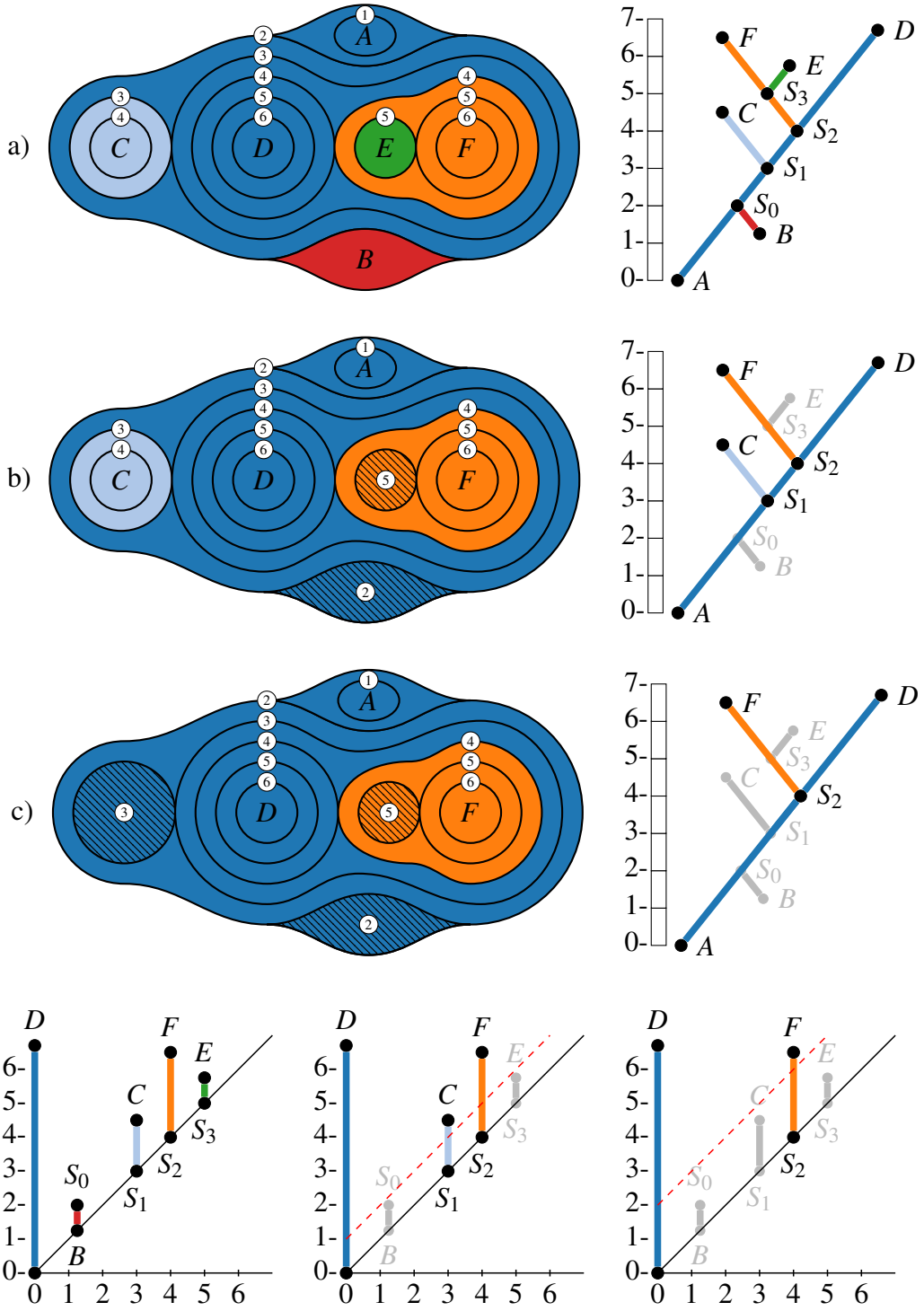


Figure 2.14: Illustration of the topological simplification of a contour tree (a-c right), and its corresponding domain segmentation (a-c left), based on thresholding the persistence of branches (bottom left-right). Filtered branches that do not exceed the persistence threshold (red dashed line) are iteratively collapsed onto preserved branches they are connected to by setting the values inside the associated domain subsets (hatched areas) to the corresponding saddle values.

Removal of Critical Points

A powerful application of persistent homology is to modify an input scalar field such that it no longer exhibits a selected set of critical points [6, 29, 30, 106]. The core concept behind this approach is to set the values of the region associated with an extremum to the value of its paired saddle. This process can be imagined by chopping of hills at saddles, which removes the extremum and the associated saddle (Fig. 2.14). Persistence pairs that correspond to branches of merge and contour trees provide an intuitive choice on which critical points to remove since less persistent branches are attached to more persistent branches, i.e., small hills emerge from larger hills. Thus, the branches $\mathcal{B}$ of a contour tree $\mathcal{C}_f$ that are below a persistence threshold (red dashed line in the bottom of Fig. 2.14) can be iteratively collapsed onto more persistent branches, which results in a series of simplified functions that lack the critical points of the filtered branches. Specifically, the branches $\mathcal{B}$ between extrema and saddles that do not exceed the persistence threshold are sorted by persistence and inserted into a queue $\bar{\mathcal{B}}$ in descending order. Each iteration $i \geq 0$ processes a branch $\bar{\mathcal{B}}_i$ with corresponding persistence pair (u,v), and as long as there exists at least one branch above the persistence threshold, it is guaranteed that $\bar{\mathcal{B}}_i$ is connected to a branch with higher persistence at the saddle $s \in \{u,v\}$ located at one of the endpoints of $\bar{\mathcal{B}}_i$. Recall, the contour tree is associated with the domain segmentation ϕ that maps any point of the domain to a point on the tree. Thus, it is possible to define a new function $\hat{f}_i$ that is identical to $\hat{f}_{i-1}$, expect that the pre-images of $\bar{\mathcal{B}}_i$ through ϕ^{-1} are mapped to the constant value $\hat{f}_{i-1}(s)$ of the saddle s, i.e.,

$$\hat{f}_i(x) = \begin{cases} \hat{f}_{i-1}(s), & \text{if } x \in \phi^{-1}(B_i) \\ \hat{f}_{i-1}(x), & \text{otherwise.} \end{cases}$$

where $\hat{f}_{-1} := f$. Note, this procedure handles the recursive collapsing of branches since they are processed in descending order. This procedure creates plateaus with the constant value of their corresponding saddles (hatched areas in Fig. 2.14). Based on the theory described in the previous sections, these plateaus can not contain critical points, and thus the contour tree computation will treat the plateaus as sets of regular points. Hence, the contour tree and the scalar field have truly been simplified. Therefore, extracted level, sublevel, and superlevel sets can now be reliably identified as valid features. To summarize, persistence-based topological simplification is a robust tool to remove noise and rank features. An efficient algorithm that can even remove an arbitrary selection of critical points in any order is described in the work of Tierny et al. [106], and is available in the *Topology ToolKit* [104].

2.4 FEATURE TRACKING

The previous sections defined for a PL scalar field a robust topology-based feature characterization that derives and simplifies the corresponding contour tree segmentation. This section now introduces feature tracking approaches that aim to correlate features across a sequence of PL scalar fields, e.g., ordered samples of a time-varying scalar field.

This work focuses on feature tracking techniques that iteratively determine the relationship between features of subsequent PL scalar fields defined on the same PL manifold. Such sequences occur frequently in practice, as numerical methods advance scalar fields in steps, and machines record measurements in discrete time intervals. This makes tracking a challenging task as the results depend heavily on the difference between the scalar fields (the temporal resolution), and on the used feature characterization (the feature stability), e.g., the more two timesteps are apart, the more uncertain is the relationship between features. Moreover, even if features of subsequent fields are likely correlated, it is always possible to construct a counter example of an intermediate timestep that contradicts the assumption. Hence, every tracking method inevitably incorporates uncertainty. Therefore, it is assumed that the fields do not vary drastically, and that all fields exhibiting significant feature evolutions are elements of the sequence.

As introduced in earlier chapters, features are often characterized via sublevel or superlevel sets. Tracking methods for such features can be roughly categorized into geometrical and topological approaches [40, 70, 86, 100]. Respective prime examples are methods based on spatial overlap [7, 12, 63, 65, 66, 92, 96, 97, 99, 114], and critical point matching [10, 23, 27, 79, 100]. For instance, the top of Fig. 2.15 illustrates a time-varying scalar field whose only maximum is moving from the left to the right side of the domain, where superlevel sets are shown via distinct colors. The chosen level directly influences the overlap-based tracking as overlaps become less likely at high levels. Algorithms that match the corresponding critical points of the superlevel sets perform better in this case, as all three critical points can be identified as single moving feature, independent of the level. In general, however, no approach always performs better. Instead, one has to evaluate the implications of the chosen strategy. Overlap-based algorithms are very sensitive to the temporal resolution and the feature geometry, but guarantee that at least parts of matched features resided at the same spatial location. However, if the temporal resolution is too low, then even overlapping features might actually be distinct entities, or features might not overlap at all. Critical point matchers are not restricted to spatial overlap and incorporate additional assumptions to match features, but their accuracy relies therefore heavily on the implementation of these assumptions.

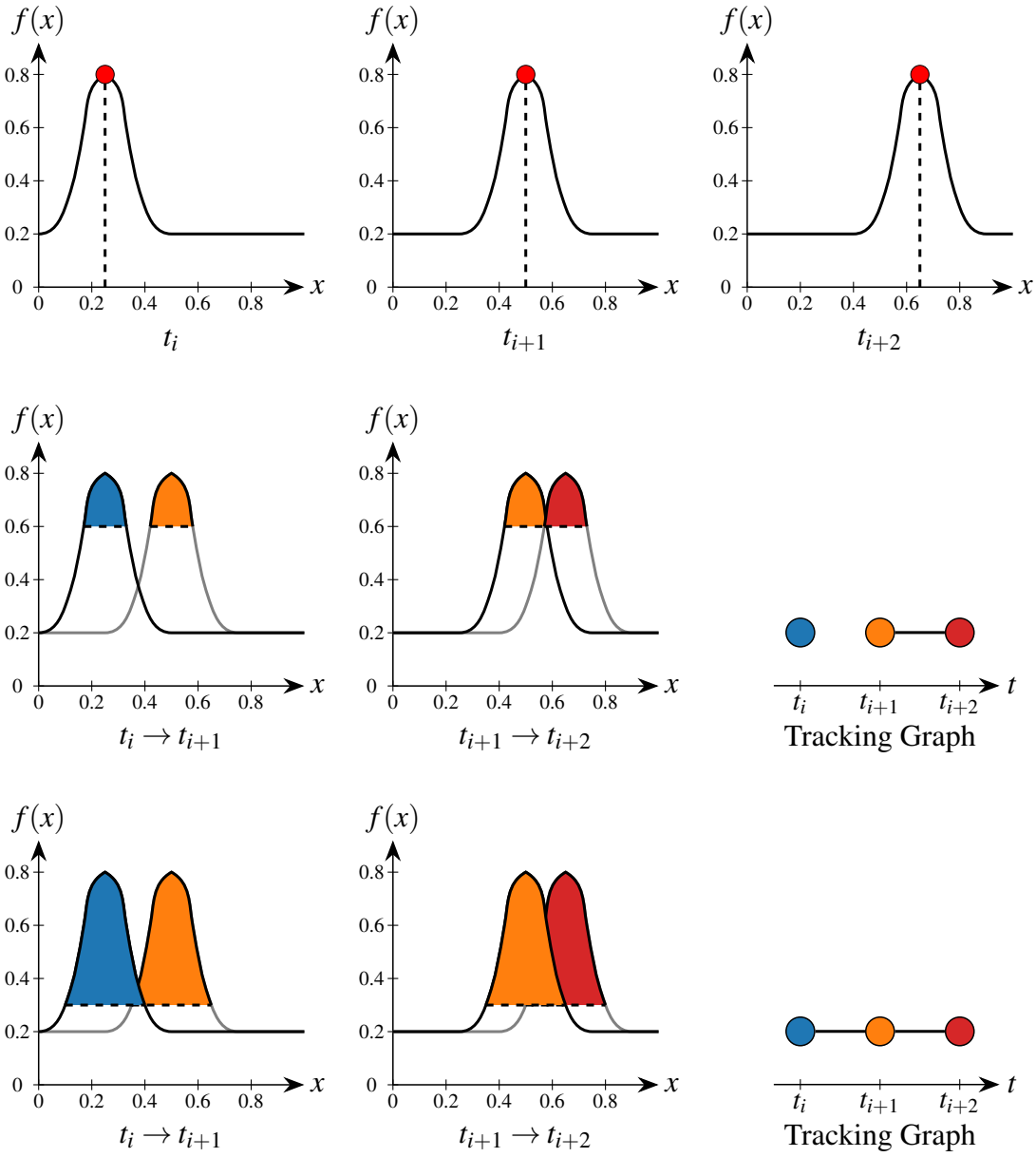


Figure 2.15: Spatial overlap-based tracking of superlevel sets (colored regions) for level 0.6 (middle) and 0.3 (bottom) of a one-dimensional time-varying scalar field (top) with a single maximum (red disc) that moves over time from the left to the right side. Note, the level has a significant impact on the feature correlation since the level impacts the volume of features, i.e., the likelihood of an overlap. Thus, features might incorrectly be identified as distinct entities (middle left). Features and their relationship over time are represented by tracking graphs (middle right and bottom right).

2.4.1 Tracking Graphs

No matter which tracking method is actually used, the resulting correlations are recorded via a topological abstraction called a tracking graph $\mathcal{T}$ (Def. 49) [12, 92, 114], i.e., a one-dimensional simplicial complex whose vertices correspond to individual features, and whose edges indicate a relationship between them. Thus, $\mathcal{T}$ represents the temporal evolution of features—i.e., when they appear, disappear, merge, and split—not unlike the contour tree, which records the evolution of features across levels instead of time. The vertices $\mathcal{V}$ of $\mathcal{T}$ are also associated with the map $\lambda : \mathcal{V} \to \mathbb{N}$ that assigns each vertex (feature) a unique label across the entire sequence, and the map $\tau : \mathcal{V} \to \mathbb{N}$ that records for a vertex the index of its corresponding scalar field inside the sequence. Vertices and edges might also store additional feature and tracking information, such as the feature size, center of mass, bounding box, or the amount of spatial overlap. Just like vertices of contour trees, vertices of $\mathcal{T}$ can be labeled as birth, death, split, merge, and regular vertices if they have no predecessor, no successor, multiple predecessors, multiple successors, or exactly one predecessor and successor, respectively. Furthermore, $\mathcal{T}$ can be decomposed into a branch decomposition $\mathcal{B}$ (Def. 43), where branches represent the evolution of individual features across multiple timesteps. Specifically, new branches emerge at birth nodes, and merging branches pass on the label of the oldest branch. If a branch splits, then its label is passed on to the branch with the longest lifespan, and the other branches are assigned new labels. Alternatively, branch labels could also be inherited based on other metrics; such as the amount of spatial overlap or the likelihood of the matching. These metrics enable again the application of persistent homology.

A common tracking graph visualization is a 2D embedding, where one axis represents time, and the other axis is solely used to arrange an optimized graph layout (Fig. 2.16 right). To follow the evolution of individual features, and to link the tracked features with the graph, both can be colored based on the feature or branch label map. The width of edges can also encode a metric such as the amount of spatial overlap or the size of features (Fig. 2.16d).

Definition 49 (Tracking Graph) *Let $\mathcal{V}$ be a set of vertices, and let the sequence map $\tau : \mathcal{V} \to \mathbb{N}$ assign to each vertex a natural number (e.g., a time index). Then, a tracking graph $\mathcal{T}$ is a one-dimensional simplicial complex consisting of the vertices $\mathcal{V}$, and any edge set $\mathcal{E}$ s.t. $\forall \langle u, v \rangle \in \mathcal{E} : \ \tau(u) = \tau(v) - 1$. $\mathcal{T}$ is also associated with the injective feature label map $\lambda : \mathcal{V} \to \mathbb{N}$ that assigns to each vertex of $\mathcal{T}$ a unique label, and $\mathcal{T}$ can be decomposed into branches $\mathcal{B}$.*

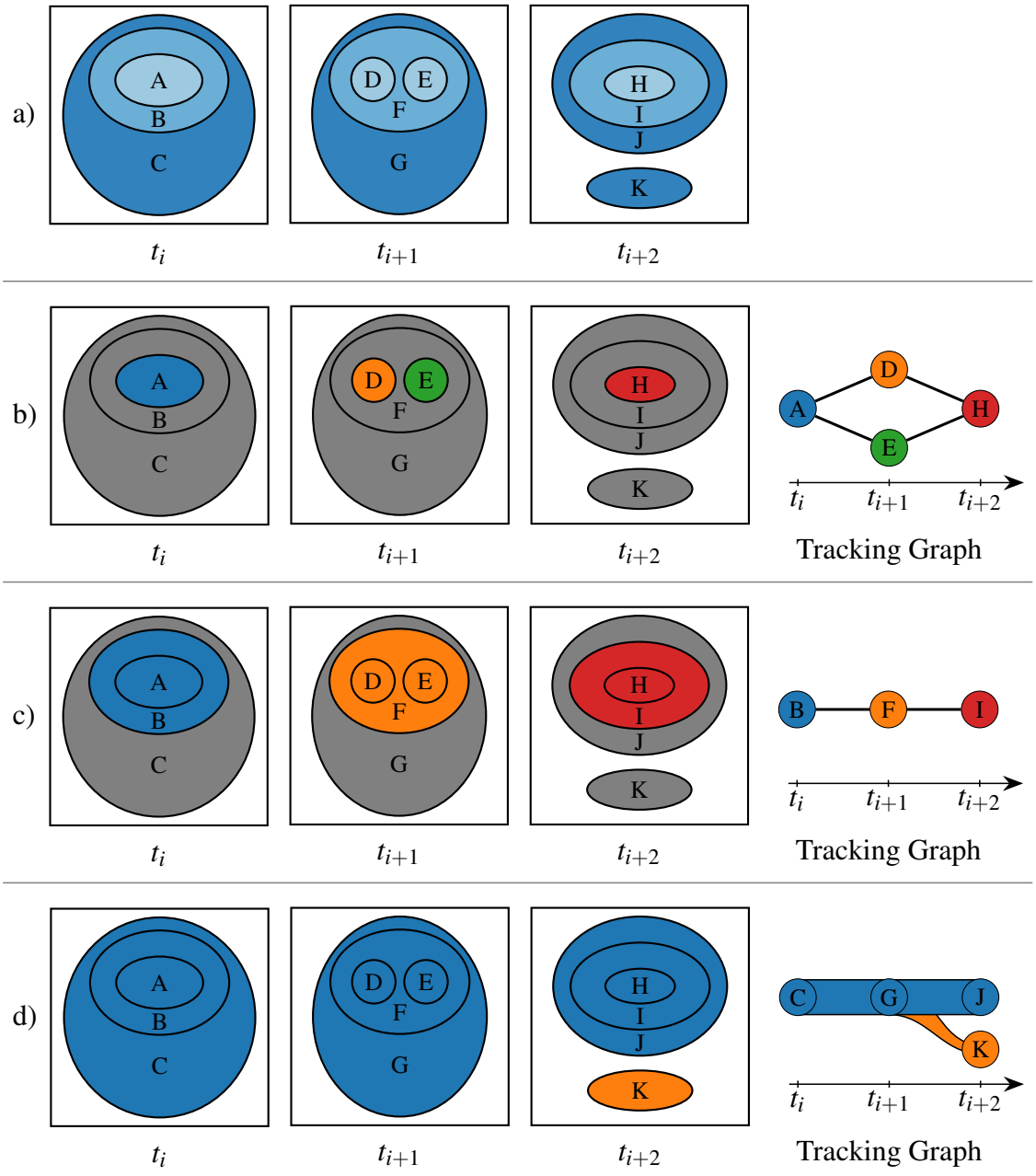


Figure 2.16: Spatial overlap-based tracking of superlevel sets (colored regions) of a two-dimensional time-varying scalar field (a) for three different levels (b-d). Figure b-c use color to represent the unique labels assigned by the feature label map λ, whereas Figure d is colored based on the branch decomposition $\mathcal{B}$. In this example, branch labels are inherited based on the largest amount of spatial overlap, which can additionally be encoded by the width of the tracking graph edges. Coloring features and the tracking graph based on the branch decomposition makes it easy to follow the evolution of individual features, and provides a link between the graph and the spatial domain.

2.4.2 Tracking via Spatial Overlap

A simple—yet effective—tracking concept is to match features whose volumes reside at least partially at the same spatial location [7, 12, 63, 65, 66, 92, 96, 97, 99]. Samtaney et al. [92] are among the first authors who used this method to track and visualize the evolution of sublevel sets in computational fluid dynamics simulations. Current tracking frameworks are still based on the same principle, such as the work of Bremer et al. [12] who track overlapping burning cells in large-scale combustion simulations, where cells are defined as areas exceeding a fuel consumption rate threshold, i.e., superlevel sets. Another example is the global tracking algorithm proposed by Saikia et al. [91] that matches regions based on the amount of spatial overlap.

In their core, these algorithms need to determine the overlap between features of subsequent scalar fields. Using the topology-based feature characterization introduced in Sec. 2.3.1, this requires computing on the same PL manifold $\mathcal{M}$ the overlap between two component segmentations $\dot{\mathcal{S}}_0$ and $\dot{\mathcal{S}}_1$ (Def. 40). Such segmentations for a PL scalar field f, a fixed level l, an initial component label n, and a mode m can be computed with the procedure *ComputeCS(f, $\mathcal{M}$, l, n, m, $\mathcal{V}$)* outlined in Alg. 1. This procedure also inserts into a vertex set $\mathcal{V}$ for each component a representative that is located at the component center, and the mode m determines if the procedure computes a sublevel or superlevel set component segmentation. For example, Fig. 2.17 illustrates two component segmentations defined on a uniform rectangular grid, where vertices are drawn as squares containing their respective labels (the background label -1 is omitted), and the representatives of the components (colored regions) are shown via red discs. Based on that feature characterization, the procedure *ComputeOverlap($\dot{\mathcal{S}}_0$, $\dot{\mathcal{S}}_1$, $\mathcal{M}$, $\mathcal{V}$, $\mathcal{E}$)* described in Alg. 3 simultaneously iterates over $\dot{\mathcal{S}}_0$ and $\dot{\mathcal{S}}_1$ to insert into a set $\mathcal{E}$ edges between the representatives of overlapping components. To determine these representatives, it is assumed that each component has a unique label in both segmentations, so that the subprocedure *getVertex($\mathcal{V}$, q)* can return the unique vertex $v \in \mathcal{V}$ that represents the component with label q. However, when components merge or split, then their respective centers "jump" to new locations. To differentiate between actual component movement and these jumps, edges of the tracking graph are either drawn as solid or dashed lines, respectively (Fig. 2.17 right). The primary tracking procedure *TrackingViaOverlap(F, $\mathcal{M}$, l, m)* of Alg. 4 then executes these subroutines iteratively for a sequence of PL scalar fields F to derive the corresponding tracking graph that consists of the resulting sets $\mathcal{V}$ and $\mathcal{E}$.

Algorithm 3: ComputeOverlap(CS $\dot{\mathcal{S}}_0$, CS $\dot{\mathcal{S}}_1$, PLM $\mathcal{M}$, Vertices $\mathcal{V}$, Edges $\mathcal{E}$)

foreach vertex $v \in \mathcal{M}$ **do**
  if $\dot{\mathcal{S}}_0[v] \geq 0 \ \wedge \ \dot{\mathcal{S}}_1[v] \geq 0$ **then**
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{ \langle \text{GetVertex}(\mathcal{V}, \dot{\mathcal{S}}_0[v]), \text{GetVertex}(\mathcal{V}, \dot{\mathcal{S}}_1[v]) \rangle \}$
return

Algorithm 4: TrackingViaOverlap(TVPLSF $\bar{F}$, PLM $\mathcal{M}$, Level l, Mode m)

$\mathcal{V} \leftarrow \emptyset$ *// Tracking Graph Vertices*
$\mathcal{E} \leftarrow \emptyset$ *// Tracking Graph Edges*
$n \leftarrow 0$ *// Component Label Counter*
// Compute First Segmentation
$\dot{\mathcal{S}}_0 \leftarrow$ ComputeCS($\hat{F}_0$, $\mathcal{M}$, l, m, n, $\mathcal{V}$)
// Compute Overlap Across Sequence
for $i \leftarrow 1$ **to** Size($\bar{F}$) **do**
  $\dot{\mathcal{S}}_1 \leftarrow$ ComputeCS($\bar{F}_i$, $\mathcal{M}$, l, m, n, $\mathcal{V}$)
  ComputeOverlap($\dot{\mathcal{S}}_0$, $\dot{\mathcal{S}}_1$, $\mathcal{V}$, $\mathcal{E}$)
  $\dot{\mathcal{S}}_0 \leftarrow \dot{\mathcal{S}}_1$
return $(\mathcal{V}, \mathcal{E})$

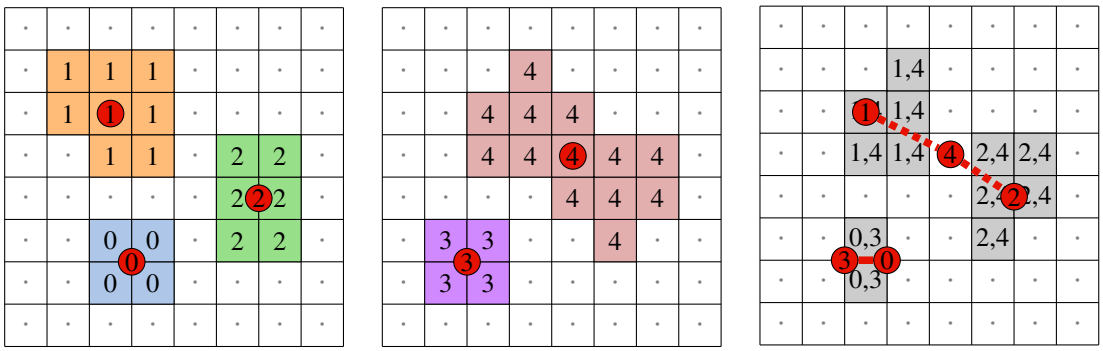


Figure 2.17: Overlap-based tracking can be efficiently computed by simultaneously iterating over two segmentations while recording concurrently present feature labels. Individual components (colored regions) are represented via new vertices located at their centers (red discs), which are connected by an edge (red lines) iff their respective components overlap. Edges that share a vertex are drawn as a dashed line in order to differentiate between feature movement and split/join events.

Tracking Hotspots in Event Datasets

Overlap-based tracking is also used in the initial work [63, 65] that later evolved to the more general and robust methodology presented in this manuscript [66]. Specifically, it has been shown that evolving disease and crime hotspots can be identified as overlapping superlevel sets of spatio-temporal kernel density estimates (KDEs) [65]. For each timestep, a two-dimensional density estimate is sampled on vertices of a uniform rectangular grid to derive a continuous PL scalar field approximation of an unstructured set of n points $P = \{e_1, \ldots, e_n\}$, where each point $e_i = ((x_i, y_i), t_i) \in P \subset \mathbb{R}^2 \times \mathbb{R}$ corresponds to an event with a spatial and temporal coordinate. Based on the underlying scenario, analysts are able to adjust the used KDE function

$$KDE(x,y,t) = \frac{1}{n\,h_t\,h_s^2} \sum_{i=1}^{n} K_T\left(\frac{t_i - t}{h_t}\right) K_S\left(\frac{x_i - x}{h_s}, \frac{y_i - y}{h_s}\right) \tag{2.3}$$

by choosing appropriate spatial and temporal kernels K_S and K_T, as well as bandwidths h_s and h_t, respectively. A common temporal kernel is a symmetric triangular function

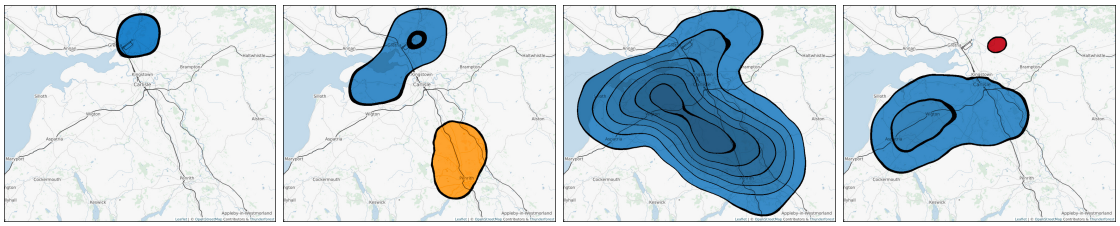
$$K_T(u) = (1 - |u|)\mathbb{1}_{\{|u|<1\}} \tag{2.4}$$

where $\mathbb{1}$ is the indicator function. This kernel incorporates the fact that the probability of a new event at time t is in some way proportional (in this case linear) to the number of other events that occur shortly before or after t. A common spatial kernel is the multivariate multiplicative Epanechnikov kernel

$$K_S(u,v) = \frac{9}{16}(1-u^2)(1-v^2)\mathbb{1}_{\{\,|u|<1 \,\wedge\, |v|<1\,\}} \tag{2.5}$$

that linearly weights events based on their distance to the sample point. The impact of events in both kernels are controlled by the spatial and temporal bandwidths, e.g., large bandwidths smooth the resulting estimate and thus limit the discriminability between hotspots, whereas small bandwidths make hotspot detection more unstable. Therefore, bandwidths have to be chosen carefully based on the underlying scenario. For instance, Lukasczyk et al. [65] chose for a foot-and-mouth disease (FMD) case study a spatial and temporal bandwidth of $t_s = 10\ km$ and $t_h = 14\ days$ to respectively incorporate the USDA suggested quarantine perimeter of infected live stock, and the FMD incubation period. Hotspots can then be identified as regions exceeding a density threshold, which can even be further filtered by persistence (Fig. 2.18).

Another advantage of the described tracking algorithm is that it simultaneously computes a spatial projection of the tracking graph by positioning its vertices at the hotspot centers (Fig. 2.19). This provides a comprehensible summary on where hotspots appear, move, merge, split, and disappear. As mentioned earlier, solid lines represent moving hotspots, whereas dashed lines indicate where hotspots join and merge, which causes a noticeable discontinuity of the hotspot centers. To improve the encoding of the temporal channel, critical events are also tagged with a timestamp, and the varying hotspot sizes are represented by the thickness of edges.



Feb 28th Mar 5th Mar 22th Apr 23th

Figure 2.18: Density estimates of foot-and-mouth disease (FMD) outbreaks in northern Cumbria county (UK) at four different dates during the 2001 FMD epidemic. Individual hotspots (colored regions) are identified as areas exceeding a minimum outbreak density threshold (outermost contour). Moreover, hotspots are consistently colored according to a branch decomposition that is derived based on the lifetime of hotspots.

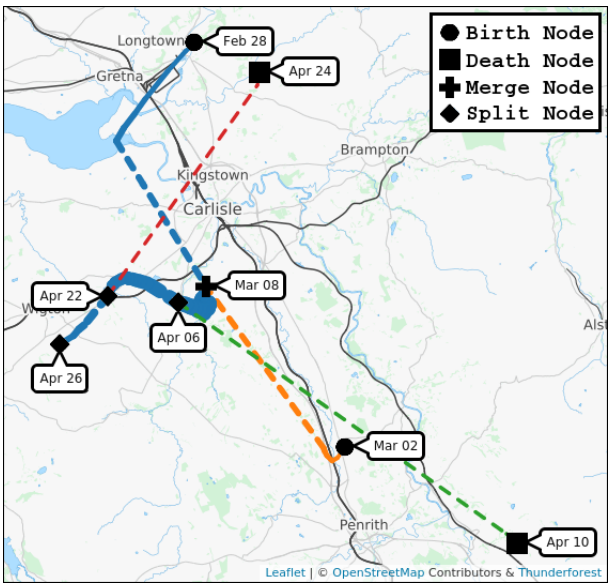


Figure 2.19: Spatial embedding of a tracking graph that represents the evolution of foot-and-mouth disease hotspots shown in Fig. 2.18. Edges are colored according to the same branch decomposition. Solid lines represent moving hotspot centers, while dashed lines indicate when hotspots merge and split. The graph clearly shows that the largest hotspot (blue) appeared in Longtown at February 28th, merged with the orange hotspot at March 8th, and then slowly regresses until the end of April when the epidemic got under control.

Tracking Viscous Fingers in FPM Simulations

Follow-up work demonstrated again that the methodology of identifying features as superlevel sets and tracking them via spatial overlap is effective in practice. For instance, the extended approach described in Lukasczyk et al. [63] characterizes the process of viscous fingering, which is prominent in many fields of science and engineering [20, 33]; including geology, hydrology, and chromatography. More precisely, viscous fingering is an instability phenomenon that occurs at the interface between two fluids of distinct viscosity—e.g., when a more viscous fluid is injected into a less viscous one—and is characterized by the formation of distinctive finger-like structures, which are called viscous fingers (colored regions in Fig. 2.21). Identifying and tracking these fingers helps to understand the mixing process as their geometrical evolution describes the penetration of the less viscous fluid.

The 2016 scientific visualization contest [42] provided an ensemble of finite pointset method (FPM) simulations that model the mixing process of salt solutions inside a water filled cylinder with an infinite salt supply at its top. Each FPM simulation consists of a set of points that are advected over time and store the salt concentration value at their current location (Fig. 2.20). As soon as the salt mixes with the water, the resulting solutions sink down to the bottom as they have a higher density than the surrounding water. Tasks of the contest included the detection of viscous fingers, and the visualization of their evolution across the simulation runs. To address these tasks, the initial approach described in Lukasczyk et al. [65] was extended to the three-dimensional setting in Lukasczyk et al. [63]. First, a pre-processing step computes a kernel density estimate—or alternatively an interpolation—of the points to derive a connected PL scalar field representation of the data. Next, the method derives superlevel sets of the resulting salt concentration scalar fields that exceed a user-specified concentration level. Removing the salt supply (dark gray regions in Fig. 2.21) from the derived superlevel sets yields several connected components that correspond to the individual viscous fingers (colored regions in Fig. 2.21). Next, tracking the fingers via spatial overlap generates tracking graphs that effectively summarize their evolution (Fig. 2.21 middle). For instance, the simulation illustrated in Fig. 2.21 exhibits one primary finger structure (orange), form which other large fingers emerge. The tracking graph also clearly indicates where completely separate fingers evolve, such as the blue finger from timestep 70 to 80 (bottom right of the tracking graph).

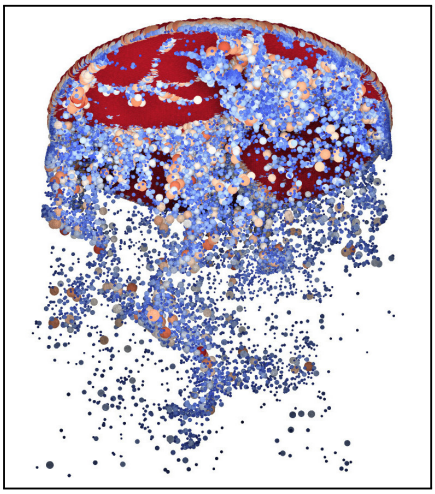


Figure 2.20: One timestep of a FPM simulation of the 2016 scientific visualization contest [42] that models the process of viscous fingering inside a water filled cylinder with an infinite salt supply at its top. The simulation advects roughly two million points that store the salt concentration value at their current location. To reduce clutter, the current frame only shows points that exceed a salt concentration threshold of 100, where concentration values are encoded by the size and color of points. Computing a 3D kernel density estimate of the points on a uniform grid yields a continuous data representation (Fig. 2.21).

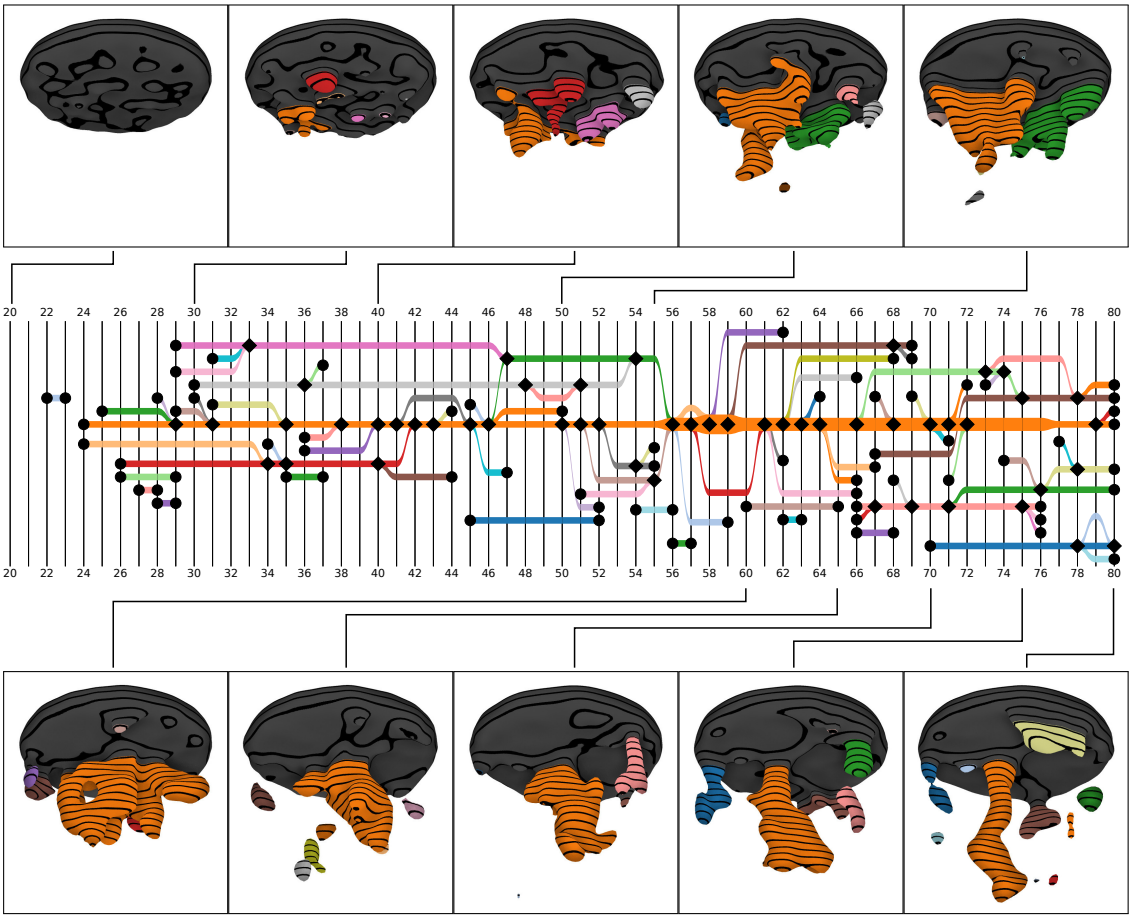


Figure 2.21: Illustration of a FPM simulation whose points have been processed by a kernel density estimator to provide a continuous data representation. Viscous fingers can then be identified and tracked via spatial overlap of superlevel sets above salt concentration level 10. As shown by the feature images (top and bottom), the branch decomposition of the tracking graph (middle) effectively illustrates when individual viscous fingers appear (discs), disappear (discs), merge (diamonds), and split (diamonds).

Instead of using the y-axis of the tracking graph solely for layout purposes, it can also encode some additional metric. For example, the y-axis of Fig. 2.22 encodes the average z-position of a finger to illustrate how salt solutions sink down to the bottom of the cylinder. Obviously, complex tracking graphs result in cluttered visualizations. To cope with this problem, fingers can be filtered by various metrics; such as persistence, size, or location. More effectively, by integrating the tracking graph and the rendering window of the fingers in a visual analytics framework enables analysts to interactively explore the graph, rotate the spatial rendering of the fingers, and change filter criteria as well as the salt concentration threshold.

Limitations

The main limitation of tracking sub- and superlevel set components based on spatial overlap is that the chosen level significantly impacts the number and shape of features, and thus the structure of the tracking graph. Updating the level also requires recomputing the tracking graph and rerendering the features. Moreover, tracking graphs for numerous features might also be extremely complex and can not be visualized without clutter. Those limitations are the primary motivations for the methodology presented in this manuscript. Specifically, Ch. 3 introduces a novel approach to characterize and visualize feature evolution across multiple levels in one compact representation, and Ch. 4 presents a strategy to store feature images in a structured database to later compose 3D views based on existing imagery instead of actually rendering features. Ch. 5 then describes a more efficient way to compute and interact with complex tracking graphs based on a combination of both approaches.

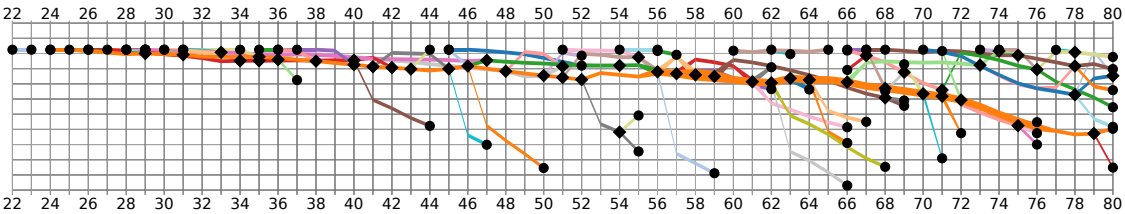


Figure 2.22: Tracking graph of Fig. 2.21 where the x-axis, the y-axis, and the line thickness encode time, average z-position of a finger, and finger volume, respectively.

2.4.3 Topology-Based Tracking Approaches

The previous section described a commonly used tracking technique based on spatial overlap, which is also used in the preliminary work that preceded the methodology described in this manuscript. Although this approach works well for the presented applications, it is not necessary true that two features are related if and only if they share some spatial overlap, e.g., if the temporal resolution is too low, then fast moving features might not overlap, or overlapping features might actually be distinct entities. This section describes techniques that correlate features based on topological abstractions, such as contour trees [12, 79, 114], persistence pairs [10, 100], Jacobi sets [23, 26, 27], Morse-Smale complexes [11, 28, 38, 53], and higher-dimensional isosurfaces [45].

An essential limitation of explicit overlap-based tracking techniques is the necessity to re-compute feature geometries every time parameters of the feature characterization change—e.g., the level for a superlevel set segmentation. Moreover, it is often infeasible to store every state of a large-scale simulation due to bandwidth and disk space constraints, which makes it impossible to explicitly re-compute feature boundaries. This triggered a line of research that aims to compute and store general tracking information during the simulation to efficiently derive tracking graphs *post hoc* without reprocessing the original data [12, 79, 114]. A prime example of such an approach is the so-called meta-graph [114] that is computed at simulation runtime to record the spatial overlap of features for discrete parameter intervals, i.e., this approach aggregates features for adjacent parameter values into groups and tracks them together. During *post hoc* analysis, meta-graphs are then used to determine the relationship between specific features by looking up the relationship of their respective groups. Hence, the accuracy of the tracking depends on the resolution of the parameter intervals. Sec. 5.2.1 describes an adaption of this approach that enables the efficient *post hoc* computation of nested tracking graphs based on merge tree segmentations.

Another strongly related tracking approach is the critical point matching algorithm proposed by Oesterling et al. [79] that actually computes the time-varying merge tree in arbitrary dimensions. Their approach is based on the fact that the structure of the merge trees only changes when the sorting order of adjacent tree vertices changes. Based on this fact, they determine a sequence of local updates that iteratively transform the merge trees over time. The resulting trees are visualized by plotting a 1D landscape profile for each timestep and connecting its peaks with lines to illustrate the evolution of all critical values across time. However, computing time-varying merge trees in this fashion is extremely expensive, and the resulting visualizations suffer from cluttering and occlusion, which makes the approach currently less suited for interactive systems and large datasets.

If features boundaries correspond to level sets for a fixed level, it is also possible to consider time as an additional dimension and compute the Reeb graph of the higher-dimensional feature domain [12, 45, 65, 112]. Edges of the Reeb graph then correspond to individual features and vertices indicate when features appear, merge, split, and disappear, i.e., in this case the Reeb graph is effectively the tracking graph. For instance, Bremer et al. [12]—and later Weber et al. [112]—analyze and track flame fronts in combustion simulations by computing the four-dimensional space-time Reeb graph.

In contrast to contour tree segmentations and Reeb graphs that partition the underlying manifold based on the evolution of level sets, the Morse-Smale complex partitions the manifold based on the gradient of a scalar field [11, 28, 38]. The central property of each Morse-Smale complex cell is that integral lines inside a cell have the same start and end point, which are critical points of the scalar field. Especially if feature boundaries coincide with the gradient, a Morse-Smale segmentation effectively separates individual features, which can subsequently be tracked by spatial overlap or other matching algorithms [53].

Recently, Soler et al. [100] proposed to compute the newly introduced lifted Wasserstein distance between persistence pairs to derive an optimal matching between features. In contrast to other global matching metrics—such as the original Wasserstein metric [46] or the Earth mover's distance [44]—the lifted Wasserstein metric also includes geometric properties of the spatial embedding of the persistence pairs to improve the tracking accuracy, and to speed up the computation. Compared to overlap-based techniques, their results indicate that this method is more stable for low temporal and spatial resolutions. However, this approach aims to establish a one-to-one relationship between features, thus, merging and splitting features have to be matched in a required post processing step. As described later in Ch. 3, the matched features do not necessarily satisfy the nesting property (Def. 50), which is why this method can not be trivially integrated into the proposed methodology. Yet, extending the method to fulfill the nesting property appears fruitful, and can be addressed in future work.

2.5 CINEMA DATABASES

The previous sections introduced common topology-based feature characterizations, such as contours and superlevel sets. At smaller scales, it is possible to compute, analyze, render, track, and store these features on demand without a significant memory footprint. For large-scale datasets and *in situ* environments, however, this is no longer the case, as it is often impractical to store entire simulation states due to bandwidth and I/O constraints. Thus, the increasing size, number, and complexity of datasets make it necessary to store a minimal amount of information that still supports effective and flexible *post hoc* data analysis and visualization.

To address this task, Ahrens et al. [2] proposed an image-based approach to store and visualize simulation output at scale via so-called Cinema databases. These databases contain color and depth images of simulation objects that were generated *in situ* based on a predefined set of camera angles, simulation parameters, and visualization operations, e.g., positions of clipping planes, timesteps, color maps, and isovalues of contours (Fig. 2.23). Each image in the database is uniquely identified by its corresponding parameter values. These databases are several orders of magnitude smaller than the simulation data they are derived from, and they enable the real-time exploration of large-scale simulations by querying and compositing images from the database (Fig. 2.24). For instance, rudimentary database viewers emulate 3D interaction with the depicted objects by snapping to the closest available camera position for a requested viewpoint [3, 63, 82, 83, 115]. The databases can also be browsed by performing queries [3], or by selecting images via parallel coordinate plots [115] or tracking graphs [63].

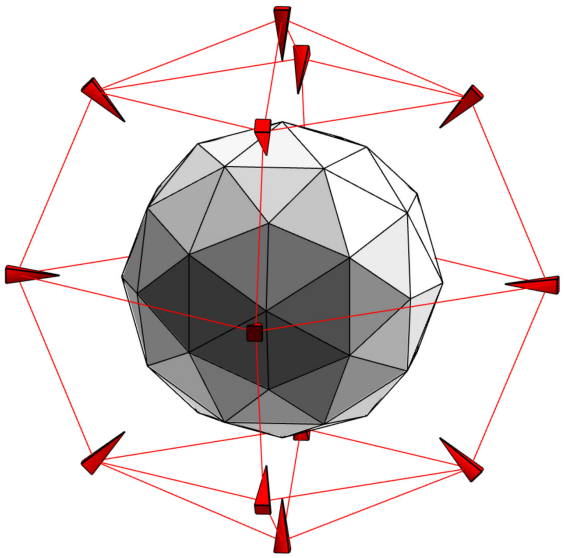


Figure 2.23: Image-sampling of a simulation object (icosahedron) using cameras (red arrows) that are located on vertices of a low-resolution spherical gird (red edges) and that aim towards the object center. A common Cinema database will then consist of one set of color images for each scalar field defined on the object, and one set of depth images.

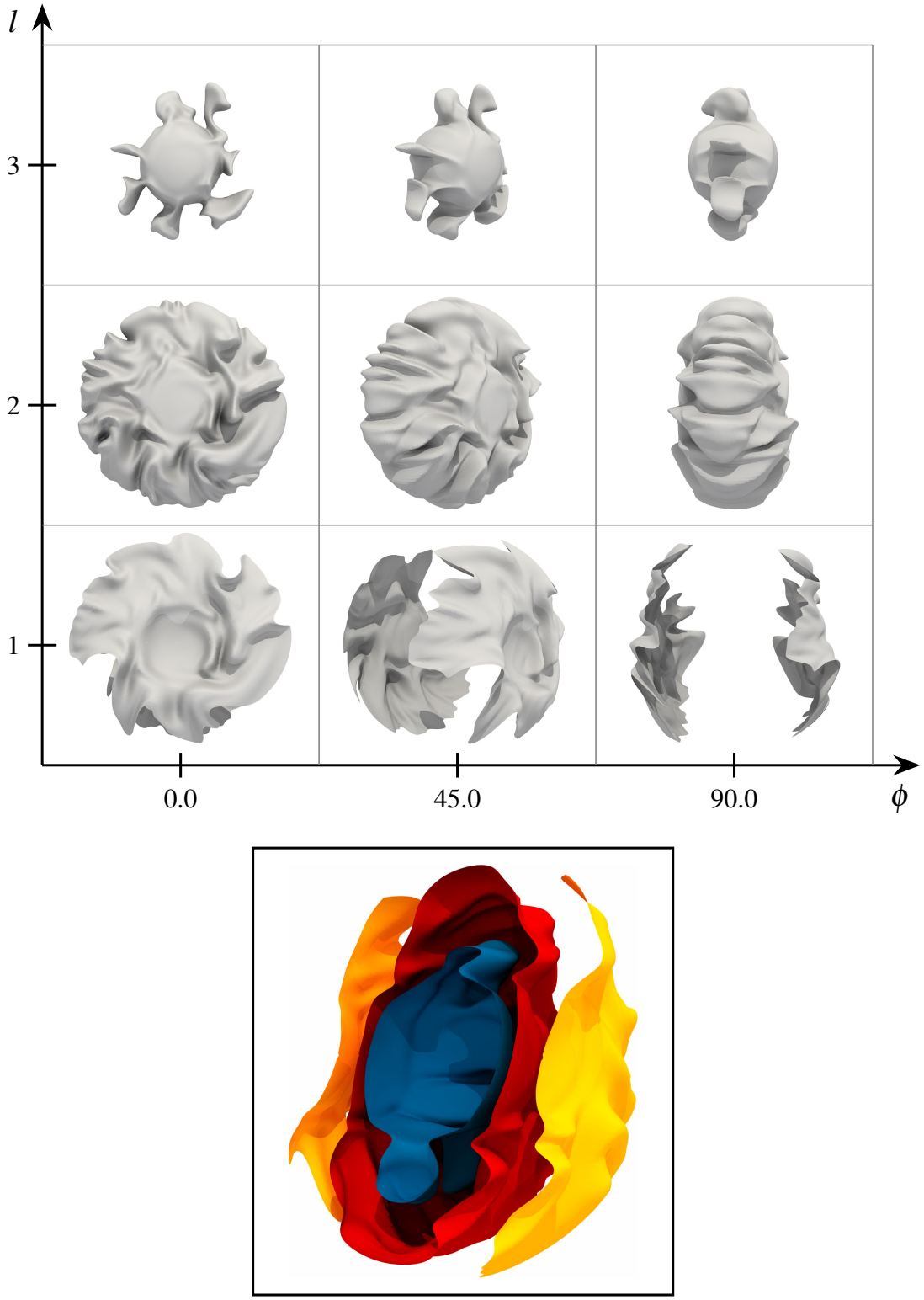


Figure 2.24: Cinema database (top) consisting of images of contours (gray surfaces) with different isovalues l (y-axis), and camera angles ϕ (x-axis). It is possible to derive new images (bottom) by compositing depth images from the database. Here, color was used to differentiate between contours at different levels.

Also due to contributions in the context of this manuscript [60, 63, 64], the original Cinema database specification has evolved to a more generic specification [90] that supports now any kind of data product, e.g., images, derived meshes, tabular data, persistence diagrams, merge trees, tracking graphs, and so forth. According to the current specification, a Cinema database is a file system folder with the extension ".cdb" that contains all data products. The associated parameters and the relative file path of each data product is stored in a comma separated value (CSV) file named "data.csv" (Fig. 2.25).

Although the Cinema specification does not prescribe how data products have to be stored, it is recommended to store them in a *Visualization ToolKit* (VTK) format [94], such as *VTKUnstructuredGrid* or *VTKImageData*. These formats represent the topology of a dataset by a set of points (i.e., vertices) and a set of cells (i.e, vertex groups), where the type prescribes cell configurations. Scalars that are associated with the points or cells are stored as so-called point or cell data, respectively. Additionally, data that is associated with the dataset as a whole—e.g., a timestep, or bins of a histogram—are stored as so-called field data. VTK formats are abstract enough to support almost any kind of data product, but also precise enough to standardize products across platforms and algorithms. For example, persistence diagrams, tracking graphs, and contours are essentially *VTKUnstructuredGrids* with additional point, cell, and field data that represent their semantic properties.

A significant limitation of Cinema databases is the fact that they are limited to the set of data products that were stored at simulation runtime. Thus, images from view angles that have not been sampled must be extrapolated from the database. For example, non-stored view angles can be approximated based on existing imagery (Sec. 2.6). Furthermore, it is possible to identify and store a limited set of view angles that will produce high quality view approximations (Ch. 4).

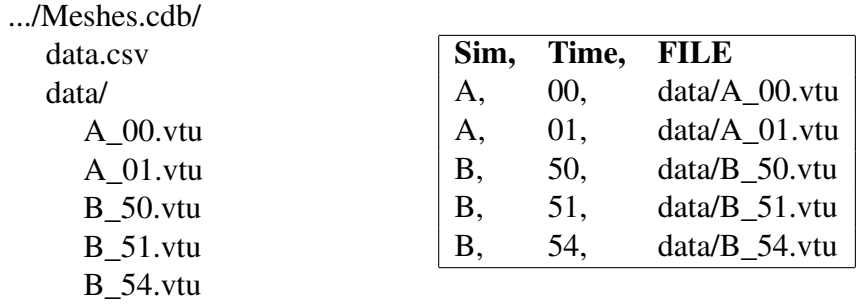


Figure 2.25: Representation of a Cinema database consisting of the "Meshes.cdb" file system folder (left) containing all related data products and a "data.csv" file (right) that records the relative path of the products and their respective simulation parameter values.

2.6 VIEW APPROXIMATION TECHNIQUES

This section introduces the principles behind view-approximation techniques that are the basis of the proposed image databases generation approach described in Ch. 4. In general, view-approximation techniques utilize existing imagery—mostly depth images—and their corresponding camera calibrations to derive synthetic images at novel view angles that minimize the visual error to the ground truth. This resulted in various Image-Based Rendering (IBR) approaches, which are summarized in the following.

IBR methods and their integrated geometry approximation algorithms have been extensively studied in the context of remote rendering [8, 19, 51], image-based meshing [21, 37, 77, 81, 116, 117], 3D video processing [75, 102], and many more. In remote rendering, they significantly reduce server and bandwidth load by enabling clients to extrapolate new views based on already transmitted images without additional requests to the server [19]. As soon as the client camera diverges too much, the server generates and sends new images to the client. This is especially useful if the visualizations require computational or data intensive procedures. In 3D video processing, they allow to *post hoc* create stereoscopic images based on video-plus-depth footage [102]. They are also used to mesh objects based on multiple photographs, which enables photorealistic texture mapping [88, 107], the digital archiving of cultural heritage [117], and the complete reconstruction of indoor as well as outdoor environments [77, 116]. Shum and Kang [95] point out that all these methods require either implicit [15, 16, 37, 57, 113, 117], explicit [19, 77, 80, 81], or no [54, 71] geometry information to create novel views based on feature registration, geometry approximation, or plenoptic functions, respectively.

2.6.1 Implicit Geometry and No-Geometry based Techniques

Implicit geometry approximation algorithms interpolate between images by detecting and tracking features—such as the optical flow [15, 57, 113] or SIFT [37]—which creates visually appealing transitions between different views. However, the interpolated images do not necessarily have to coincide with reality, and often exhibit ghosting and warping artifacts [101]. IBR algorithms that use no geometry information interpret large dense sets of images as two-dimensional slices of the four-dimensional light field function [54, 71]. All images are used to approximate the light field which is subsequently sampled to generate novel views. This produces high quality results as long as the light field approximation is good enough, but this requires a huge amount of images (1k+) and even compressed representations do not scale for non-static scenes [54].

2.6.2 Depth Image Based Rendering Techniques

On the other side of the IBR spectrum are algorithms that explicitly derive the implied geometry of depicted objects based on depth images. These images can be obtained from sensors [77, 116], estimators [39, 55, 56, 58], or directly from the rendering pipeline [2]. As it is straight-forward to generate Cinema databases that contain precise depth images of 3D rendered objects—such as isosurfaces, streamlines, and particles—the methodology presented in this manuscript focuses on Depth Image Based Rendering (DIBR) techniques. This does not mean, however, that the other approaches are unsuitable for image extrapolation from Cinema databases, which can be examined in future work.

The rest of this section provides an overview across the development of DIBR techniques, which is also the basis of the approach described in Ch. 4. These methods are based on the fact that each pixel of a depth image corresponds to a 3D point on the depicted surface (Fig. 2.26a and b). These points can be computed by inverting the projection that was used to generate the depth image [19, 77, 80, 81], which yields a set of independent points in 3D space. A simple way to render the resulting locations is to represent them as a point cloud, called splatting [78, 81, 98, 121] (Fig. 2.26b). However, this creates gaps between points; especially when the depth image has a low resolution. The gaps can be filled by increasing the point size (which leads to a strong divergence from the original surface), or by increasing the point number (which requires high-resolution depth images).

Another way to solve this problem is to bridge these gaps with linear surface approximations. To this end, it is necessary to link neighboring points of the depth image by creating a surface patch between them, i.e., to derive a triangulation based on the depth image. As a first step, one can create two triangles between four neighboring pixels to create a piecewise linear approximation of the surface between the points (Fig. 2.26c). This fills all gaps, but also creates surface patches between pixels with very different depth values. This is known in the DIBR literature as the depth discontinuity [75, 102, 120]. A trivial solution to this problem is to use a distance threshold to discard distorted triangles (Fig. 2.26d). Unfortunately, there exists no threshold value that will always produce the best results as this value strongly depends on the smoothness of the depicted object. Moreover, removing such triangles creates gaps again that must be either filled by a variant of splatting [75], or by incorporating the implied geometry from multiple depth images [16, 19, 77] (Fig. 2.27). The described DIBR algorithm is implemented in the *DepthImageBasedGeometryApproximation* module [62] of the *Topology ToolKit* [104].

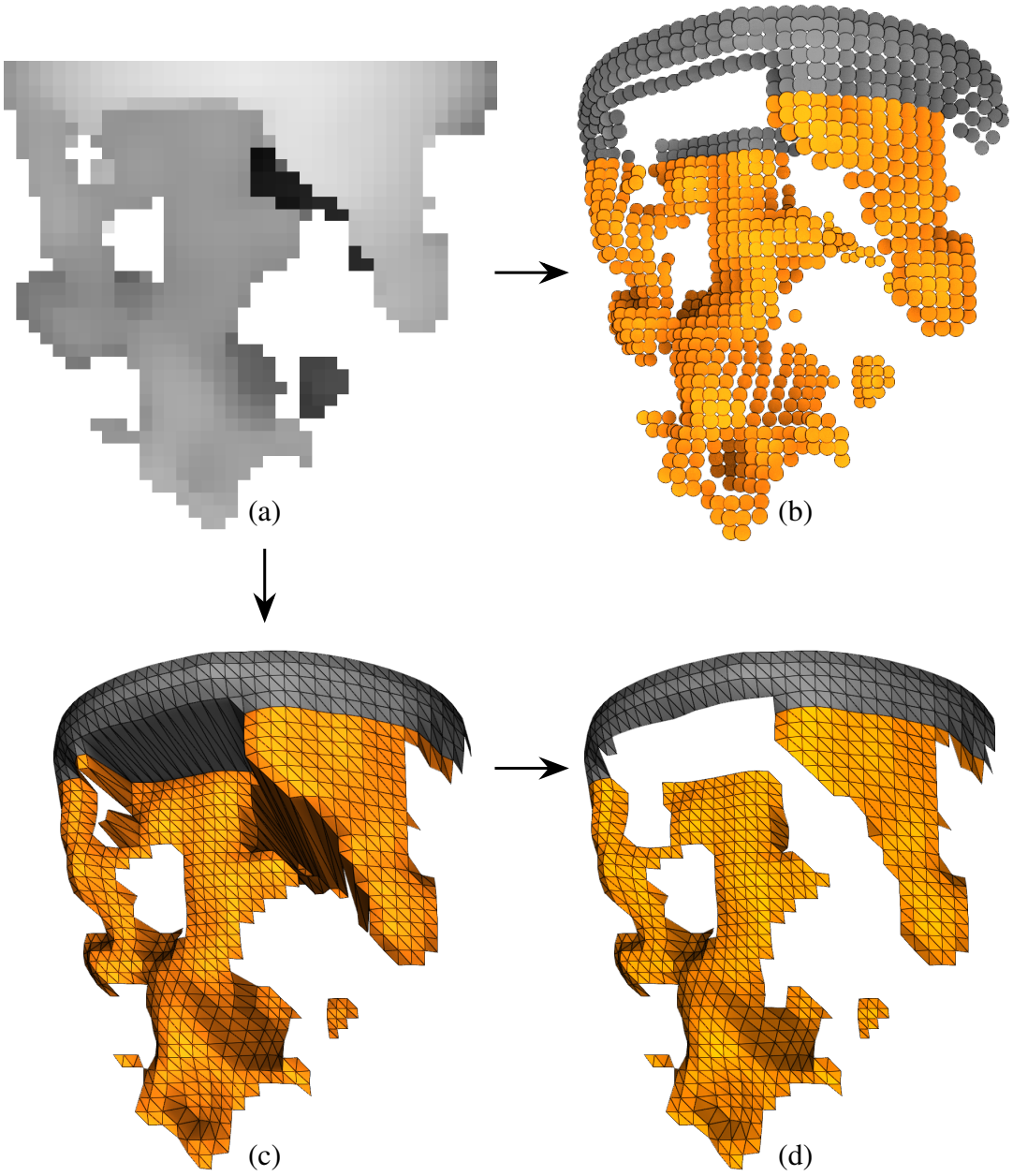


Figure 2.26: Illustration of the forward mapping of a single 40x40 depth image (a) for the viscous finger dataset. The resulting points can either be directly visualized by splatting (b), or by approximating the surface between the points (c-d). Splatting (b) creates gaps that need to be filled either by increasing the point number (i.e., the image resolution) or the point size. The surface approximation (c) creates a continuous set of piecewise linear patches between vertices, but a distance threshold is needed to discard distorted triangles (d).

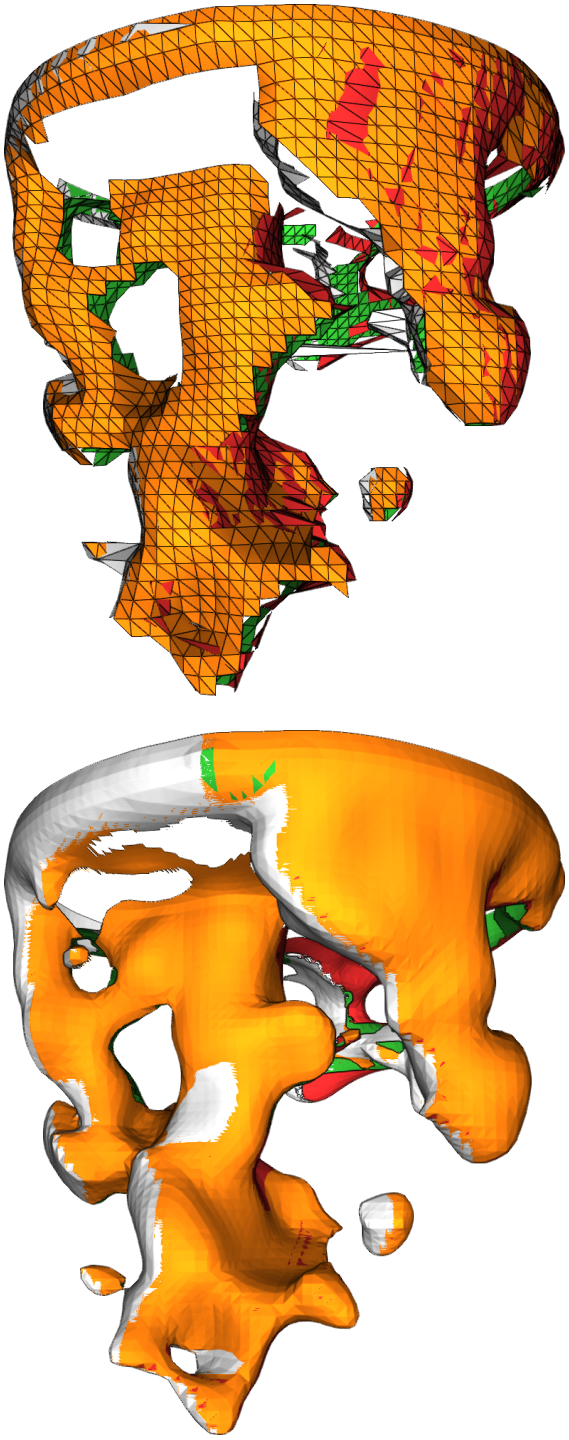


Figure 2.27: Composited surface approximations of the viscous finger dataset using four depth images with a resolution of either 40^2 pixels (top) or 1024^2 pixels (bottom). Colors encode the depth image that generated the corresponding surface patch. Depth images that depict the same part of a surface generate similar patches which causes z-fighting. This is advantageous in this case as the different depth images agree on the shape of the corresponding surface patch.

CHAPTER 3

NESTED TRACKING GRAPHS

Common tracking graphs are a well established tool in topological analysis to visualize the evolution of features and their properties over time, i.e., when superlevel and sublevel set components appear, disappear, merge, and split (Sec. 2.4). However, tracking graphs are limited to a single level threshold and the graphs may vary substantially even under small changes to the threshold (Sec. 3.1). This chapter presents a novel topological abstraction, called the nested tracking graph (NTG) [66], that records the evolution of features that exhibit a nesting hierarchy; such as the nesting hierarchy of superlevel set components for different levels (Sec. 3.2). A NTG sets multiple tracking graphs in context to each other by simultaneously illustrating feature evolution at all hierarchy levels in one compact visualization. The effectiveness of this approach is demonstrated on various time-varying datasets from computational fluid dynamics and cosmology simulations (Sec. 3.3). Based on these results, it was shown that NTGs effectively summarize feature evolution and enable interactive exploration (Sec. 3.4).

3.1 MOTIVATION

In various applications, features can be characterized as superlevel or sublevel set components of scalar fields, i.e., connected subsets of the domain whose corresponding scalars are respectively above or below a certain level threshold (Sec. 2.3). The temporal evolution of these features can the be determined through tracking approaches, such as techniques based on spatial overlap or merge tree segmentations (Sec. 2.4). The features and their evolution are then recorded by a topological abstraction called a tracking graph $\mathcal{T}$ (Def. 49). Each vertex of $\mathcal{T}$ represents an individual feature—e.g., a single superlevel set component for a specified level at a certain timestep—and edges between vertices represent a relationship between them—e.g., if one component is a descendant of component from an earlier timestep. A common way to visualize these tracking graphs is a planar embedding, where one axis is used to represent time, and the other to optimize the graph layout (Fig. 2.16 right). Thus, tracking graphs provide a comprehensible visualization of the evolution of features, i.e., when they appear, disappear, merge, and split. However, tracking graphs have in general the following limitations:

- tracking graphs can only represent one level, which is not always known a priori;
- to examine multiple levels one has to compare multiple tracking graphs; and
- tracking graphs may vary substantially even under small changes to the levels.

These limitations are illustrated in Fig. 2.16. Specifically, each tracking graph records a different story based on the chosen level, and all of these stories are relevant to understand the temporal evolution of the underlying scalar field. Therefore, the approach described in this chapter aims to derive a compact visual representation of all these tracking graphs via a so-called nested tracking graph (NTG). NTGs utilize the fact that superlevel and sublevel sets for different levels are nested inside each other, which yields a hierarchy. The top of Fig. 3.1 illustrates the three tracking graphs of Fig. 2.16 via different shades of blue, and the nesting hierarchy of the superlevel sets for each timestep via edges between their corresponding vertices in different shades of red. Based on the nesting hierarchy, it is possible to draw edges of the different tracking graphs inside each other, which summarizes their story in one graph (Fig. 3.1 middle). This visualization enables users to effectively follow the evolution of features for different levels simultaneously, while also setting edges of different levels in context to each other. For huge amounts of features, NTGs become extremely complex. To counteract this problem, Sec. 3.3 describes how to integrate NTGs as dynamic and interactive control devices in visual analytic frameworks. Linked to a 3D rendering of the original data, NTGs can be used to navigate through time and toggle the visibility of features, which enables users to perform temporal and spatial data peeling.

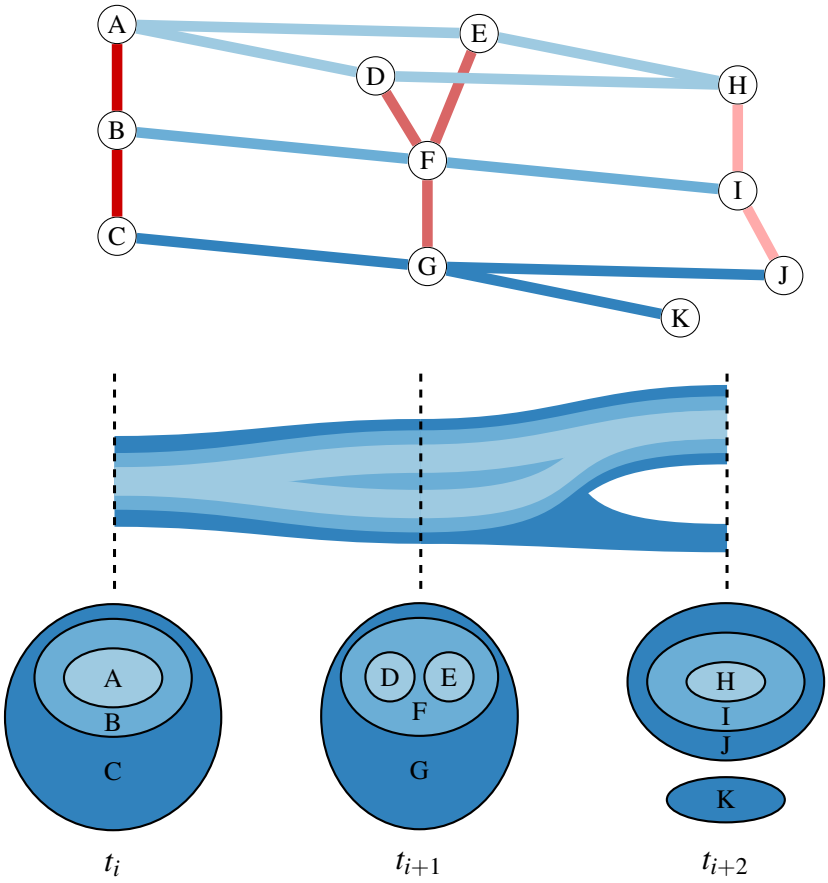


Figure 3.1: (Bottom) Superlevel set components of the time-varying scalar-field of Fig. 2.16 for three different levels (dark to light blue). (Top) 3D illustration of a nested tracking graph where the tracking graphs for each level are shown in shades of blue, and the nesting trees for each timestep in shades of red. (Middle) Nested tracking graph where edges of the tracking graphs are drawn inside each other according to the nesting trees.

3.2 APPROACH

This section introduces a formal definition of nested tracking graphs (Sec. 3.2.1), provides a rudimentary NTG computation algorithm based on spatial overlap tracking (Sec. 3.2.2), and describes a visualization algorithm for NTGs (Sec. 3.2.3).

3.2.1 Formalization

Def. 50 provides a formal description of a nested tracking graph $\mathcal{N} = \mathcal{V} \cup \mathcal{E}$, which is basically a one-dimensional simplicial complex whose vertices $\mathcal{V}$ have a sequence index τ and hierarchy index η, and whose edges $\mathcal{E}$ have to fulfill some criteria based on these indicies (Eq. 3.1–3.4). Specifically, the edge set $\mathcal{E}$ is the union of two disjoint edge sets $\mathcal{E}_T$ and $\mathcal{E}_N$ that are called the tracking graph edges and nesting tree edges, respectively. Each vertex $v \in \mathcal{V}$ that is not at the lowest hierarchy level must have a parent vertex $\hat{\eta}(v)$ with the same sequence index at hierarchy level $\eta(v) - 1$ (Eq. 3.1), and the edge set $\mathcal{E}_N$ consists of all these child-parent relationships (Eq. 3.2). Note, each complete set of edges inside $\mathcal{E}_N$ that only connect vertices with the same sequence index constitutes a so-called nesting tree or a forest of these trees (red edges of Fig. 3.1). Next, Eq. 3.3 ensures that tracking graph edges only connect vertices with adjacent sequence indicies at the same hierarchy level, and all edges of $\mathcal{E}_T$ between vertices at the same hierarchy level constitute a complete tracking graph. Finally, Eq. 3.4 requires that if two children are connected by an edge in $\mathcal{E}_T$, then their parents must also be connected by an edge in $\mathcal{E}_T$. Thus, the last criterion ensures that all tracking graph edges are nested across the hierarchy levels, and it is therefore referred to as the nesting property. Note, not every tracking algorithm naturally satisfies the nesting property. The sections that describe the two tracking algorithms of the proposed methodology (Sec. 3.2.2 and Sec. 5.3.1) also provide a proof that the computed graphs are indeed NTGs. Similar to common tracking graphs, the vertices of NTGs can be uniquely identified through a feature label map λ, and the edges can be decomposed into a set of branches $\mathcal{B}$ (Def. 43).

The described NTG definition is abstract enough to represent any kind of sequence and hierarchy relationship between vertices. In this manuscript, NTGs are primarily used to record the temporal evolution of superlevel sets and their nesting hierarchy across different levels. Yet, as demonstrated in Sec. 3.3.4, they can also be used to record the evolution of clique communities inside weighted graphs, where the nesting hierarchy is determined trough the dimension of the cliques, and the sequence results form a filtration of the graph based on the edge weights.

Definition 50 (Nested Tracking Graph) *Let $\mathcal{V}$ be a set of vertices, and let the sequence map $\tau : \mathcal{V} \to \mathbb{N}$ and the hierarchy map $\eta : \mathcal{V} \to \mathbb{N}$ assign to each vertex a natural number (e.g., a time or level index). Thus, let $\hat{\mathcal{V}} = \{v \in \mathcal{V} \mid \eta(v) > 0\}$ be the set of vertices that are not at the lowest hierarchy level, and let the parent map $\hat{\eta} : \hat{\mathcal{V}} \to \mathcal{V}$ assign to each of these vertices a parent s.t.*

$$\forall\, v \in \hat{\mathcal{V}}\, [\, \tau(\hat{\eta}(v)) = \tau(v) \;\wedge\; \eta(\hat{\eta}(v)) = \eta(v) - 1\,]. \tag{3.1}$$

A nested tracking graph $\mathcal{N}$ is a one-dimensional simplicial complex consisting of the vertices $\mathcal{V}$, and any edge set $\mathcal{E} = \mathcal{E}_T \cup \mathcal{E}_N$ s.t.

$$\mathcal{E}_N = \{\, \langle \eta(v), v \rangle \mid v \in \hat{\mathcal{V}} \,\}; \tag{3.2}$$

$$\forall\, \langle u, v \rangle \in \mathcal{E}_T\, [\, \tau(u) = \tau(v) - 1 \;\wedge\; \eta(u) = \eta(v)\,]; \textit{ and} \tag{3.3}$$

$$\forall\, \langle u, v \rangle \in \mathcal{E}_T\, [\, u, v \in \hat{\mathcal{V}} \Rightarrow \langle \hat{\eta}(u), \hat{\eta}(v) \rangle \in \mathcal{E}_T\,]. \tag{3.4}$$

$\mathcal{N}$ is also associated with the injective feature label map $\lambda : \mathcal{V} \to \mathbb{N}$ that assigns to each vertex of $\mathcal{N}$ a unique integer label, and $\mathcal{N}$ can be decomposed into branches $\mathcal{B}$.

3.2.2 NTG Computation Via Spatial Overlap

This section presents the rudimentary NTG computation algorithm that is described in Lukasczyk et al. [66], and provides a proof that the computed graph is indeed a nested tracking graph. The algorithm explicitly determines the overlap of superlevel or sublevel sets between adjacent timesteps at the same level (to determine the tracking graphs), and between adjacent levels at the same timestep (to determine the nesting trees).

Alg. 5 provides the pseudocode of the procedure *NestedTrackingViaOverlap(F, $\mathcal{M}$, L, m)* that processes for an enumeration of levels L and a mode m an enumeration of PL scalar fields F that are defined on the same PL manifold $\mathcal{M}$. Depending on whether the mode m is set to 1 or -1, the algorithm tracks sublevel or superlevel set components, respectively. The levels in L must also be sorted in descending or ascending order, accordingly. This procedure derives a set of vertics V, tracking graph edges E_T, and nesting tree edges E_N, which are all initialized as empty sets. To this end, the procedure utilizes the two subprocedures *ComputeCS* (Alg. 1) and *ComputeOverlap* (Alg. 3) to respectively derive component segmentations, and the overlap between segmentations of adjacent timesteps and levels. In a nutshell, the subprocedure *ComputeCS(f, $\mathcal{M}$, l, m, n, V)* derives either a sublevel or superlevel set component segmentation for a level $l \in L$ based on the mode m, assigns to each component a unique integer label starting at n, and inserts for every component a representing vertex into the set V. The subprocedure *ComputeOverlap(S_0, S_1, V, E)* processes two such segmentations and inserts into the set E an edge $\langle u, v \rangle$ between the representatives $u, v \in V$ of overlapping components. This function is used to derive the tracking graph edges by computing the overlap between segmentations of adjacent timesteps at the same level (line 21), and the nesting tree edges by computing the overlap between segmentations of adjacent levels at the same time index (line 11 and 18). Specifically, the algorithm iterates over the scalar fields and stores the segmentations of the previous and current iteration in the enumerations P and C, respectively. To properly iterate over the scalar fields, it is necessary to initially compute the segmentations and the corresponding nesting tree of the first timestep (line 7-11) outside the main loop (line 12-23). The last step of each iteration replaces the enumeration P with C. Finally, the algorithm returns the sets V, E_T, and E_N. Note, all edges of E_T between vertices with the same level yield a complete tracking graph, and all edges of E_N between vertices with the same time index yield a nesting tree or a forest of nesting trees. However, this rudimentary algorithm needs to recompute the components and their overlap each time the levels are updated. A more efficient algorithm is presented later in Sec. 5.3.1.

Algorithm 5: NestedTrackingViaOverlap(TVPLSF $\hat{F}$, PLM $\mathcal{M}$, Levels L, Mode m)

```
1  V   ← ∅  // Tracking Graph Vertices
2  E_T ← ∅  // Tracking Graph Edges
3  E_N ← ∅  // Nesting Tree Edges
4  n   ← 0  // Component Label Counter
5  P   ← [] // Previous Segmentations
6  C   ← [] // Current Segmentations
7  // Compute Segmentations and Nesting Tree Edges of F_0
8  for j ← 0 to Size(L) do
9      P_j ← ComputeCS( F_0, M, L_j, m, n, V )
10     if j > 0 then
11         ComputeOverlap( P_{j-1}, P_j, V, E_N )
12 // Iterate over Sequence
13 for i ← 1 to Size(F̂) do
14     // Compute Segmentations and Nesting Tree Edges of F̂_i
15     for j ← 0 to Size(L) do
16         C_j ← ComputeCS( F̂_i, M, L_j, m, n, V )
17         if j > 0 then
18             ComputeOverlap( C_{j-1}, C_j, V, E_N )
19     // Compute Tracking Graph Edges between F̂_{i-1} and F̂_i
20     for j ← 0 to Size(L) do
21         ComputeOverlap( P_j, C_j, V, E_T )
22     // Replace Previous with Current Segmentations
23     P ← C
24 return (V, E_T, E_N)
```

To proof that the computed graph of Alg. 5 is indeed a nested tracking graph, it is necessary to show that the constraints in Eq. 3.1–3.4 of Def. 50 are satisfied. Therefore, let V, E_T, and E_N correspond to $\mathcal{V}$, $\mathcal{E}_T$, and $\mathcal{E}_N$, respectively. From Alg. 5 follows directly that each component of every timestep and sampled level is represented by a unique vertex in V, and that V only consists of these representatives. To simplify notations, let $v_i^j \in V$ denote the vertex that uniquely represents a connected component at time index i and level index j. Then, let the maps τ and η return for each vertex $v_i^j \in V$ respectively the time and level index, and let the parent map $\hat{\eta}$ return for every vertex $v_i^j \in V$ with $j > 0$ the unique vertex $v_i^{j-1} \in V$ whose corresponding component completely contains the component of v_i^j. That there exists exactly one such component is proven in Theorem 1.

Theorem 1 (Superlevel and Sublevel Set Components have a Nesting Relationship) *For a PL scalar field f and two levels $a, b \in \mathbb{R}$ with $a \leq b$, each superlevel set component of $\mathcal{L}_f^+(b)$ is the subset of exactly one superlevel set component of $\mathcal{L}_f^+(a)$, and each sublevel set component of $\mathcal{L}_f^-(a)$ is a subset of exactly one sublevel set component of $\mathcal{L}_f^-(b)$.*

Proof: *Let the sets A and B denote the superlevel sets $\mathcal{L}_f^+(a)$ and $\mathcal{L}_f^+(b)$, respectively. Since for each component $\dot{B}$ of B we know that $\forall x \in \dot{B} : \; a \leq b \leq f(x)$ it follows that*

$$\dot{B} \subseteq A. \tag{3.5}$$

Let $\bar{A}$ denote an enumeration of all connected components of A, i.e., $A = \bigcup \bar{A}$. Hence, Eq. 3.5 implies that $\dot{B}$ is at least a subset of one component of $\bar{A}$. Lets assume that $\dot{B}$ is the subset of more than one component of $\bar{A}$, and let $\dot{A}_0, \dot{A}_1 \in \bar{A}$ be any two distinct components that have a non-empty intersection with $\dot{B}$. Now consider the points $x \in \{A_0 \cap \dot{B}\}$ and $y \in \{A_1 \cap \dot{B}\}$. Since x and y are elements of the same connected component $\dot{B}$ there must exist a path $P \subseteq \dot{B}$ from x to y. However, from Eq. 3.5 follows that

$$P \subseteq \dot{B} \subseteq A \tag{3.6}$$

which means that $\dot{A}_0$ and $\dot{A}_1$ are connected by a path in A. This contradicts the fact that $\dot{A}_0$ and $\dot{A}_1$ are distinct connected components of A. Hence, $\dot{B}$ is a subset of exactly one component of A. The proof for sublevel set components is symmetrical. □

Using the aforementioned definitions of $\mathcal{V}$, $\mathcal{E}_T$, $\mathcal{E}_N$, τ, η, and $\hat{\eta}$, and the fact that Alg. 5 only adds to E_N edges between vertices that have the same time index and that are at adjacent levels (lines 11 and 18), it follows that Eq. 3.1 and 3.2 of Def. 50 hold. Eq. 3.3 holds as line 21 of Alg. 5 only adds edges to E_T at the same hierarchy level for adjacent timesteps. The nesting property (Eq. 3.4) follows form Eq. 3.5, i.e., if the components of two vertices v_i^j and v_{i+1}^j of V with $j > 0$ overlap, than also do the components of the parent vertices $\hat{\eta}(v_i^j)$ and $\hat{\eta}(v_{i+1}^j)$ of V that contain these components. Thus, the edges $\langle v_i^j, v_{i+1}^j \rangle$ and $\langle \hat{\eta}(v_i^j), \hat{\eta}(v_{i+1}^j) \rangle$ are elements of E_T. This completes the proof that the graph computed by Alg. 5 is a valid nested tracking graph.

3.2.3 Visualization

This section describes a visualization algorithm for nested tracking graphs (NTGs), and how NTGs can be integrated in visual analytic interfaces for interactive exploration.

A Layout Algorithm for Nested Tracking Graphs

A straight forward way to visualize an NTG is to first compute individually for each level an optimized layout of the corresponding tracking graph, then draw the lowest level, and finally nest the remaining graphs inside each other from the lowest to the highest level. To this end, the procedure *ComputeNestedTrackingGraphLayout($\mathcal{N}$)* of Alg. 6 derives for a NTG $\mathcal{N} = (\mathcal{V}, \mathcal{E}_T, \mathcal{E}_N)$ a map $p : \mathcal{V} \to R^2$ that assigns to each vertex v a position in $\mathbb{R}^2$, where one coordinate represents time, and the other is used to minimize the number of edge crossings. To nest edges, it is necessary to also assign a width $w(v)$ to each vertex $v \in \mathcal{V}$ such that $w(v) \geq \sum_{u \in \hat{\eta}^{-1}(v)} w(u)$, i.e., each parent is larger than all its children. For example, the width can encode the total number of children of a parent, or the size of the associated superlevel set component. First, the algorithm computes individually for each level of $\mathcal{N}$ an optimized layout of the corresponding tracking graph $\mathcal{T}_i = \{ \langle u, v \rangle \in \mathcal{E}_T \mid \eta(u) = \eta(v) = i \} \cup \{ v \in \mathcal{V} \mid \eta(v) = i \}$ with the subprocedure *ComputeTrackingGraphLayout*. For example, this subprocedure could represent $\mathcal{T}_i$ in the graph description language DOT, and subsequently compute an optimized layout with a graph library such as Graphviz [32]. Next, the algorithm iterates over all levels above zero in ascending order, and updates the positions of the vertices at level l (children) based on the vertices at the previous level $l-1$ (parents) via the subprocedure *GetRelativePosition*.

Algorithm 6: ComputeNestedTrackingGraphLayout(NTG $\mathcal{N}$)

```
1  p ← [ ]  // Vertex Positions
2  // Compute Tracking Graph Layouts
3  for i ← 0 to NumberOfLevels(N) do
4  |   T_i ← GetTrackingGraph( N, i )
5  |   ComputeTrackingGraphLayout( T_i, p )
6  // Update Vertex Positions of Nested Levels
7  for i ← 1 to NumberOfLevels(N) do
8  |   T_i ← GetTrackingGraph( N, i )
9  |   foreach vertex v ∈ T_i do
10 |   |   p[v] ← GetRelativePosition( N, p, v )
11 return p
```

A child vertex v is connected via exactly one edge of $\mathcal{E}_N$ to its parent $u = \hat{\eta}(v)$ due to Eq. 3.1 and 3.2 of Def. 50. The new location of v depends on the number and width of all other children of its parent $\hat{\eta}^{-1}(u)$ (Figure 3.2). As the total width of all children does not exceed the width of the parent, children can be drawn below each other inside the available space of the parent, where the order depends on the optimized layout calculated for the tracking graph $\mathcal{T}_j$. The remaining space of the parent can then be used to create gaps. After the iteration, each vertex $v \in \mathcal{V}$ has a new layout coordinate $p(v)$. To actually render a NTG using the computed layout, every tracking graph edge $\langle u, v \rangle \in \mathcal{E}_T$ can be drawn as a Bézier curve from $p(u)$ to $p(v)$, where the width is linearly interpolated between $w(u)$ and $w(v)$, and edges are sorted in z-direction based on the level. Although the color scheme used to encode the different levels can be domain specific, in general it appears sensible to use a sequential color map.

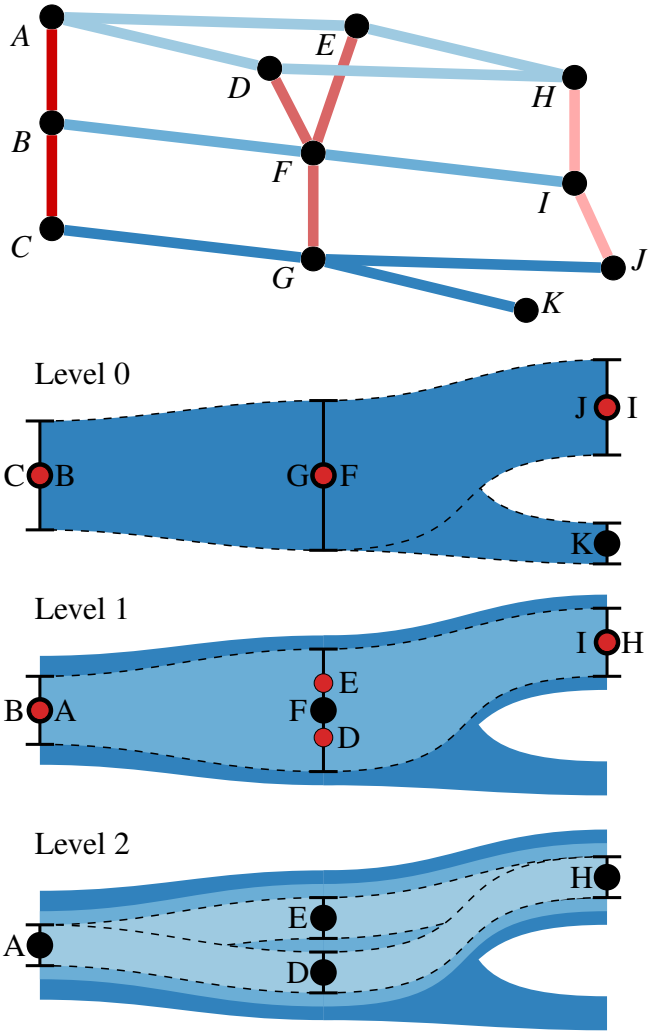


Figure 3.2: Illustration of the NTG layout algorithm. First, the algorithm computes an optimal layout for each individual tracking graph of a nested graph (blue edges of top figure), where each vertex has also an associated width. Then, the algorithm positions the vertices of the lowest layer according to this layout, and then iterates over the remaining levels in ascending order, where children (red nodes with label on the right) are positioned according to their parents (black nodes with label on the left) and the available space of the parent (black bars). After all vertex positions of one level have been determined, edges of that level can be drawn via Bézier curves that linearly interpolate the width of vertices (dashed lines). Each level is drawn in a different color to highlight the nesting hierarchy.

Integration of NTGs in Visual Analytic Interfaces

Nested tracking graphs can be integrated in visual analytic frameworks as interactive devices that illustrate the evolution of features, and enable analysts to browse through time and levels. Figure Fig. 3.3 shows a simple web-based tool that consists of a direct volume rendering (DVR) window (top), and a nested tracking graph (bottom). In this example, the NTG represents the evolution of superlevel set components for three different levels. Per default, each level of the nested graph is shown in a different color to provide an overview across the different levels. Clicking on an edge of the NTG highlights the corresponding level while other levels are grayed out, which also updates the shown components in the DVR window. The resulting highlighted graph is a common tracking graph that is colored based on a branch decomposition. Although other levels are grayed out, they still provide context as they indicate the nesting hierarchy with respect to the selected level. For instance, the nested graph in Figure Fig. 3.3 shows that 1) the huge orange component contains multiple components of higher value, 2) all components of the selected level are contained in one single component of lower level, and 3) sometimes small components split from this low level component. Components in the DVR and NTG window are shown in the same color to link both views, and it is possible to highlight individual features in both views by clicking on components in the DVR window, or by selecting vertices, edges, or entire branches of the NTG.

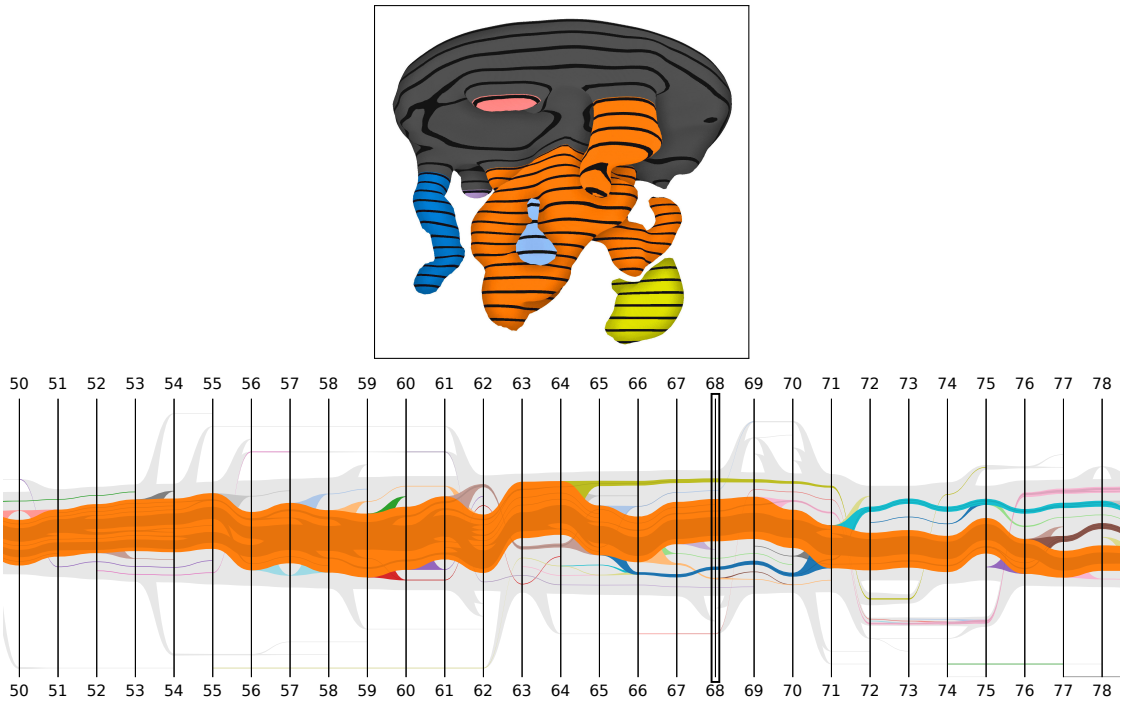


Figure 3.3: Interface of a simple web-based visual analytics framework consisting of a DVR window (top) and the interactive nested tracking graph (bottom).

3.3 RESULTS

This section demonstrates how nested tracking graphs can be used for ensemble comparison, semantic decomposition, and interactive exploration.

3.3.1 Viscous Fingering

The first case study examines the ensemble of finite pointset method (FPM) simulations of the scientific visualization contest 2016 [42] that was already introduced in Sec. 2.4.2. To summarize, the simulations model the mixing process of salt solutions inside a water filled cylinder with an infinite salt supply at its top, where simulations incorporate stochastic effects to model the aleatoric uncertainty of the mixing process. While the solutions sink down to the bottom of the cylinder, they form characteristic structures with increased salt concentration value, called viscous fingers (Fig. 3.3 left). Utilizing the approach presented in Lukasczyk et al. [63], it is possible to sample the salt concentration density on a uniform grid with 64^3 vertices, and subsequently compute superlevel set components that exceed a fixed salt concentration threshold. Removing the salt supply form these components—e.g., by clipping the domain—produces a new set of connected components that correspond to the individual viscous fingers (Fig. 3.3 left).

The original approach presented in Lukasczyk et al. [63] computes a tracking graph for multiple density levels (Fig. 2.21). Although each tracking graph effectively summarizes the finger evolution at a fixed level, they are limited to said level, and there is no direct visual link between them. Conversely, NTGs set these tracking graphs in context to each other, i.e., they illustrate how fingers of different concentration levels are nested inside each other, and they provide a compact visual representation of each run. In previous approaches, users had to compare multiple separate tracking graphs per ensemble member. Thus, computing for each run an NTG for the same set of levels enables the effective visual comparison between ensemble members. Fig. 3.4 shows the NTGs for three ensemble members (left to right) for the density levels 25, 30, and 35 (dark to light red), where the width of edges encodes the size of their associated components. Obviously, the stochastic effects of the simulation have an impact on the finger structures, yet, some trends become apparent. For instance, until around timestep 30, small fingers emerge from the salt supply that subsequently merge into one huge component shown in dark red. In all runs there exists only one of these huge growing components that sometimes splits into—or merges with—much smaller components. Furthermore, the NTGs clearly show that the number of fingers and their nesting hierarchy are very similar across the runs.

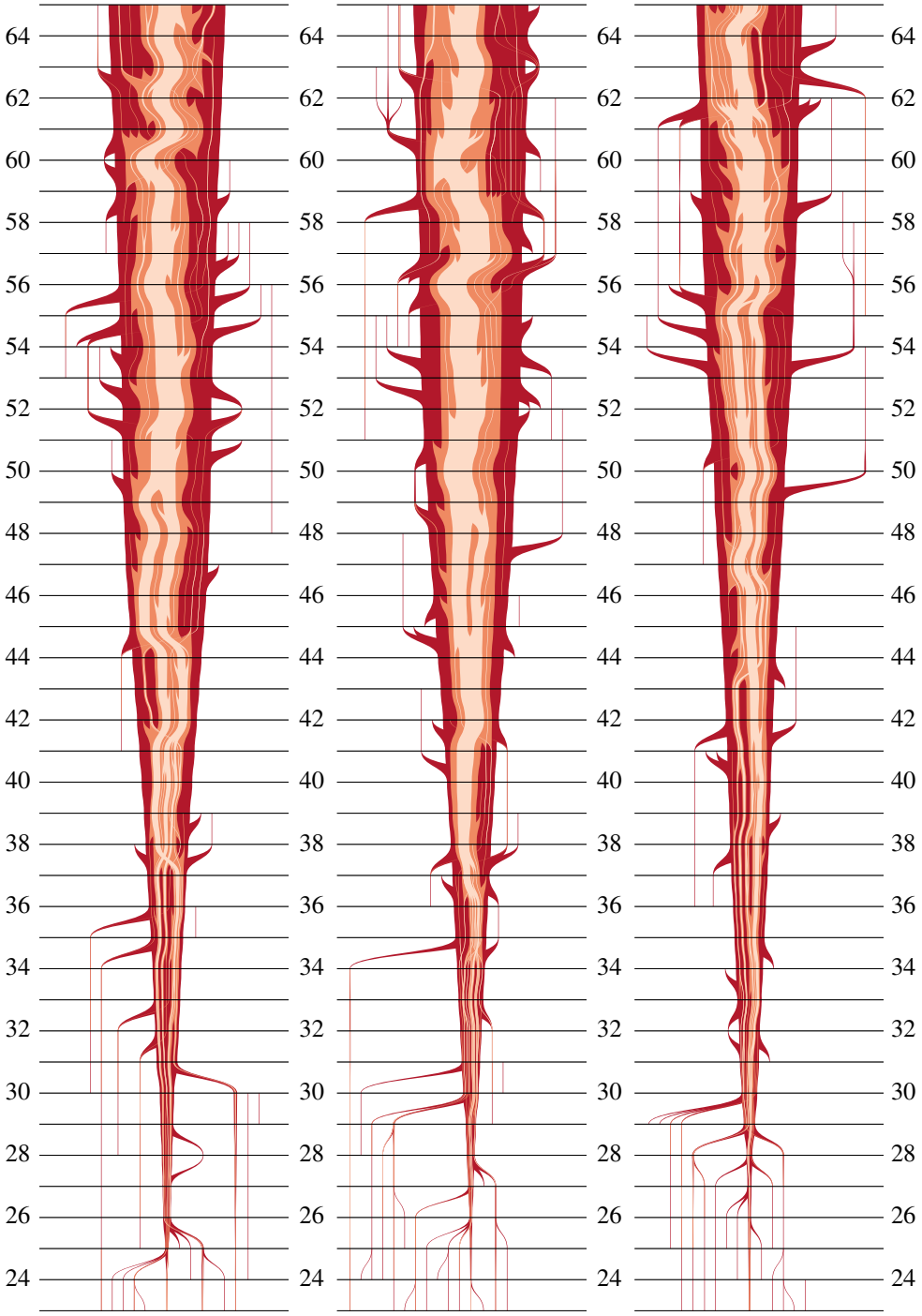


Figure 3.4: NTGs of three ensemble members of the viscous finger dataset (left to right) for the density levels 25, 30, and 35 (dark to light red). Edges represent the evolution of finger volumes, where the y-axis represents time, and the x-axis is used to minimize the number of edge crossings. Although stochastic effects alter simulation results, the graphs show similar trends such as the initial phase where small fingers originate from the salt supply and then merge into larger finger structures.

3.3.2 Jet Simulation

This dataset results from a direct, numerical, computational fluid dynamics (CFD) simulation capturing the injection of a jet into a medium at rest, which causes the formation of vortical structures due to friction. At the beginning of the simulation it can be observed that a large vortex is formed at the tip of the jet, and that this vortex progressive decays into smaller vortical structures as the system moves towards turbulence. Specifically, the simulation computes velocity data on a uniform grid of resolution $128 \times 256 \times 128$ with a total of 600 timesteps, where the derived vorticity magnitude describes the local strength of rotation. Thus, vortices of the vector field correspond to superlevel set components above a vorticity magnitude threshold.

This case study demonstrates that NTGs can be used to create semantic partitions of datasets, helping users to effectively peal through the data. For example, consider the components for the two vorticity magnitude levels 85 and 117 shown at the top of Fig. 3.5. A standard tracking graph that illustrates the evolution of the individual components at level 117 primarily consists of numerous separate lines, which is therefore heavily cluttered and does not provide context. However, the components of level 117 are contained in components of lower levels, which yields a group hierarchy that can be illustrated via a nested graph. The bottom of Fig. 3.5 shows the NTG for these two values, where the layer for level 85 is highlighted and the layer for level 117 is grayed out. The colors of the graph match the ones used to show the individual components for level 85 of top left DVR window. At timestep 302 there exist two major components, i.e., the main jet (red) and the top ring (orange). These components contain the smaller components with higher vorticity magnitude and thus provide context by partitioning them into groups. The graph shows that the ring—and thus its subcomponents—split from the main jet at timestep 295; an information that is not conveyed by conventional tracking graphs. Furthermore, if users want to examine the components within the ring, they can click on its corresponding edge to highlight the history of its subcomponents and filter out others. Thus, the nested tracking graph can be used to organize multiple tracking graphs and their respective components.

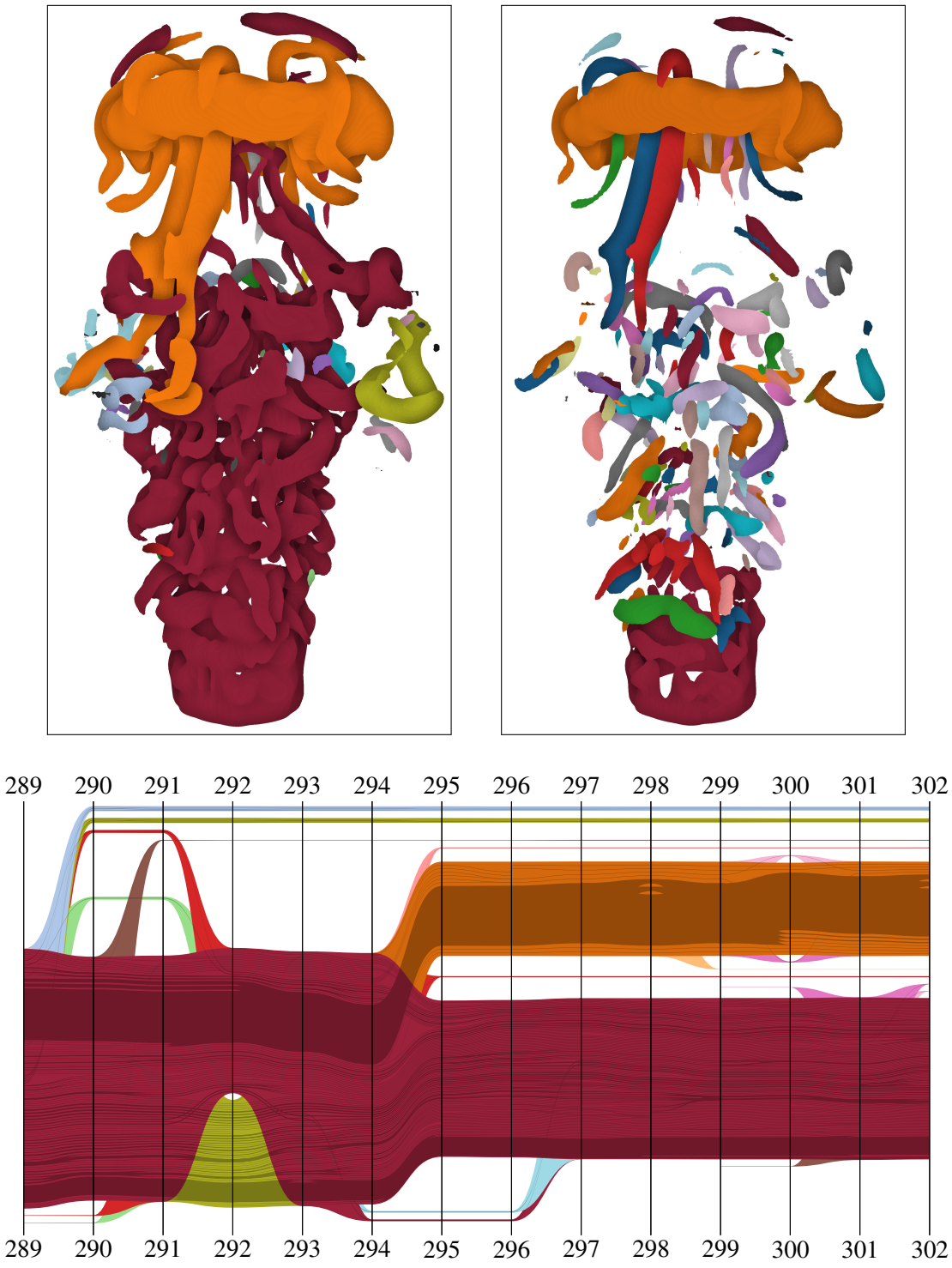


Figure 3.5: (Top) Individual components of the jet dataset at timestep 302 for vorticity magnitude level 85 (left) and 117 (right). (Bottom) NTG with focus on layer 85, i.e., layer 117 is grayed out and the edge colors of layer 85 match the components of the left DVR window. The graph indicates that the top component (orange) splits from the main component (red) at timestep 295, and contains another huge component.

3.3.3 Dark Matter Halos

In this case study, NTGs are used to visualize the evolution of dark matter halos in a large-scale cosmology simulation of the Lyman α forest [67]. The simulation is based on the Nyx [4] code, covers a cubic domain consisting of 256^3 vertices with an edge length of approximately 93 million light years, and contains 850 timesteps that span the interval from redshift $z = 159$ (approximately 10 million years after the Big Bang) to redshift $z = 0$ (today, approximately 13.5 billion years later). It uses hydrodynamics to evolve Baryon density and treats dark matter as collisionless particles evolved via a particle-mesh method. Next, halos—i.e., gravitationally collapsed regions of locally higher density—are identified as superlevel sets exceeding a density threshold, and correspond to clumps of matter hosting galaxies and groups of galaxies.

This dataset is challenging due to the vast number of features and their complex evolution; especially at the beginning of the simulation where halos start to form and then progressively cluster together. The evolution of galaxy filaments, halos, and sub-halos can be illustrated by an NTG with density levels 2.633×10^{12}, 1×10^{12}, and 5×10^{11}, respectively. The largest level was suggested by domain scientists, and the other levels are chosen heuristically based on the indicated structures of the cosmic web that are visible in the volume rendered images of the halo dataset. Specifically, the middle of Fig. 3.6 illustrates the nesting relationship of sub-halos (red), halos (dark blue), and galaxy filaments (light blue). This hierarchy can be well represented with nested tracking graphs (top and bottom). However, it is not possible to interactively render the entire graph due to the large number of edges. Therefore, the graph can be interactively explored in a level-of-detail approach by filtering halos below a certain size, collapsing intermediate timesteps, and focusing on individual feature groups. The top and bottom of Fig. 3.6 show the NTG for the same galaxy filament during the early and late stages of the simulation, respectively. The first graph shows an important phase of the simulation in which a large number of new halos are born within the same filament, that are then attracted to each other and merge. Note the vast number of small isolated halos (thin red lines) at timestep 300 that merge into two large clusters (thick red lines) until timestep 350. Later, the simulation converges to a state where most halos are clustered together. A feature in the filament that is preserved during the entire simulation run is a single huge halo (thick dark blue edge) that contains the most and largest sub-halos (red). In contrast to single tracking graphs, the nested representation shows the evolution of galaxy filaments, contained halos, and how they cluster together simultaneously.

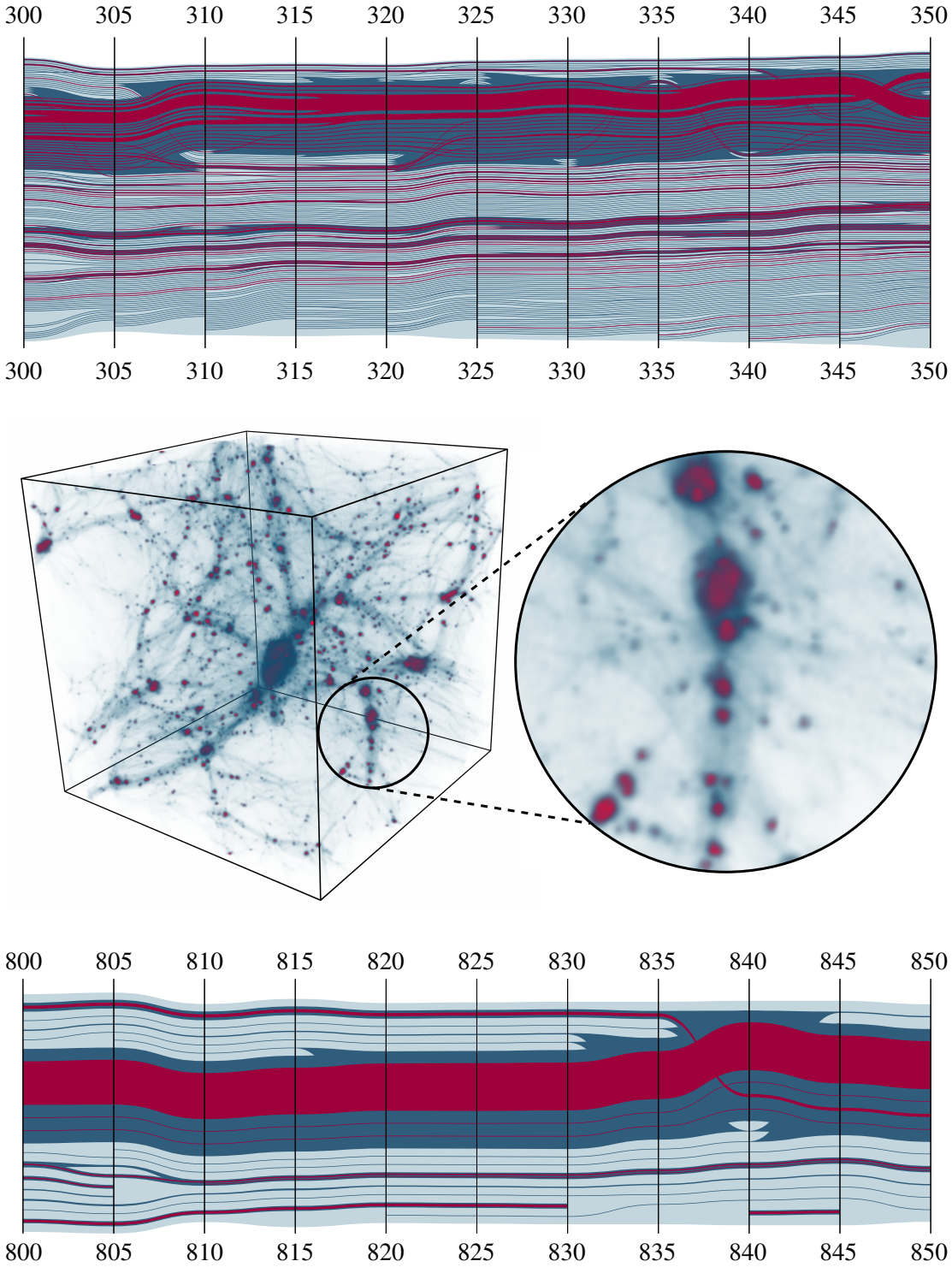


Figure 3.6: NTGs for one galaxy filament of the halo dataset during an early (top) and a late (bottom) stage of the simulation that illustrate the evolution of superlevel set components for density levels 2.633×10^{12}, 1×10^{12}, and 5×10^{11}, which correspond to galaxy filaments (light blue), halos (dark blue), and sub-halos (red). The middle shows a DVR image of timestep 850, where the transfer function matches the colors of the NTG.

3.3.4 Clique Communities

This last case study demonstrates that NTGs can also be used to illustrate the evolution of other features that exhibit a nesting hierarchy. Recently, Rieck et al. [89] proposed a novel method to analyze complex networks based on the persistence of so called clique communities, which correspond to nested densely interconnected subgraphs. Formally, the method processes a network that is represented as a one-dimensional simplicial complex $\mathcal{K}_w$ whose edges are associated with a weight value w. A k-clique is then a completely interconnected subgraph of $\mathcal{K}$ consisting of k vertices; and two k-clicks are called adjacent iff they share a $k-1$ face. A k-clique community $\mathcal{C} \subseteq \mathcal{K}$ is a maximal set of k-cliques that can be decomposed into a sequence such that every consecutive pair is adjacent, where k is called the community degree. Thus, 2-clique communities consist of edge connected components, 3-click communities are triangles that are connected by edges, and so forth. These groups decompose the entire graph, with the additional advantage that simplicies can be part of multiple communities. Therefore, the relevance of simplicies can be measured by the number and degrees of the communities they participate in. Note, this yields a nesting hierarchy as by definition every k-clique community is a subset of exactly one $k-1$-clique community, e.g., a 3-clique community is also an edge connected component and so forth. Moreover, defining a filtration of $\mathcal{K}_w$ based on thresholding w yields a sequence of growing graphs $\bar{\mathcal{K}}_w$. Thus, the evolution of communities in $\bar{\mathcal{K}}_w$ for multiple degrees can be described by a nested tracking graph.

Fig. 3.7 illustrates the well-known co-occurrence network between characters in Victor Hugo's novel "Les Misérables". To derive a filtration from significant to insignificant connections between characters, it is first necessary to invert the edge weights. Thus, characters that frequently interact with each other are connected by an edge with a small weight. The network only consists of 77 vertices and 254 edges, but contains numerous cliques up to $k = 10$. The edges of the NTG (top) comprehensibly illustrate the evolution of k-clique communities during the filtration, where the x-axis and colors correspond to the weight threshold and community degrees, respectively. For small edge weight values, the network consists of a single small 2-clique community. By increasing the weight threshold, new 2-clique communities start to appear and merge. At the same time, colors turn into brighter shades, thereby revealing that the connections between vertices become stronger. In contrast to a standard connectivity analysis, the use of clique communities reveals the presence of various communities. In this case, although all characters participate in the same story, there exists numerous subplots involving different characters.

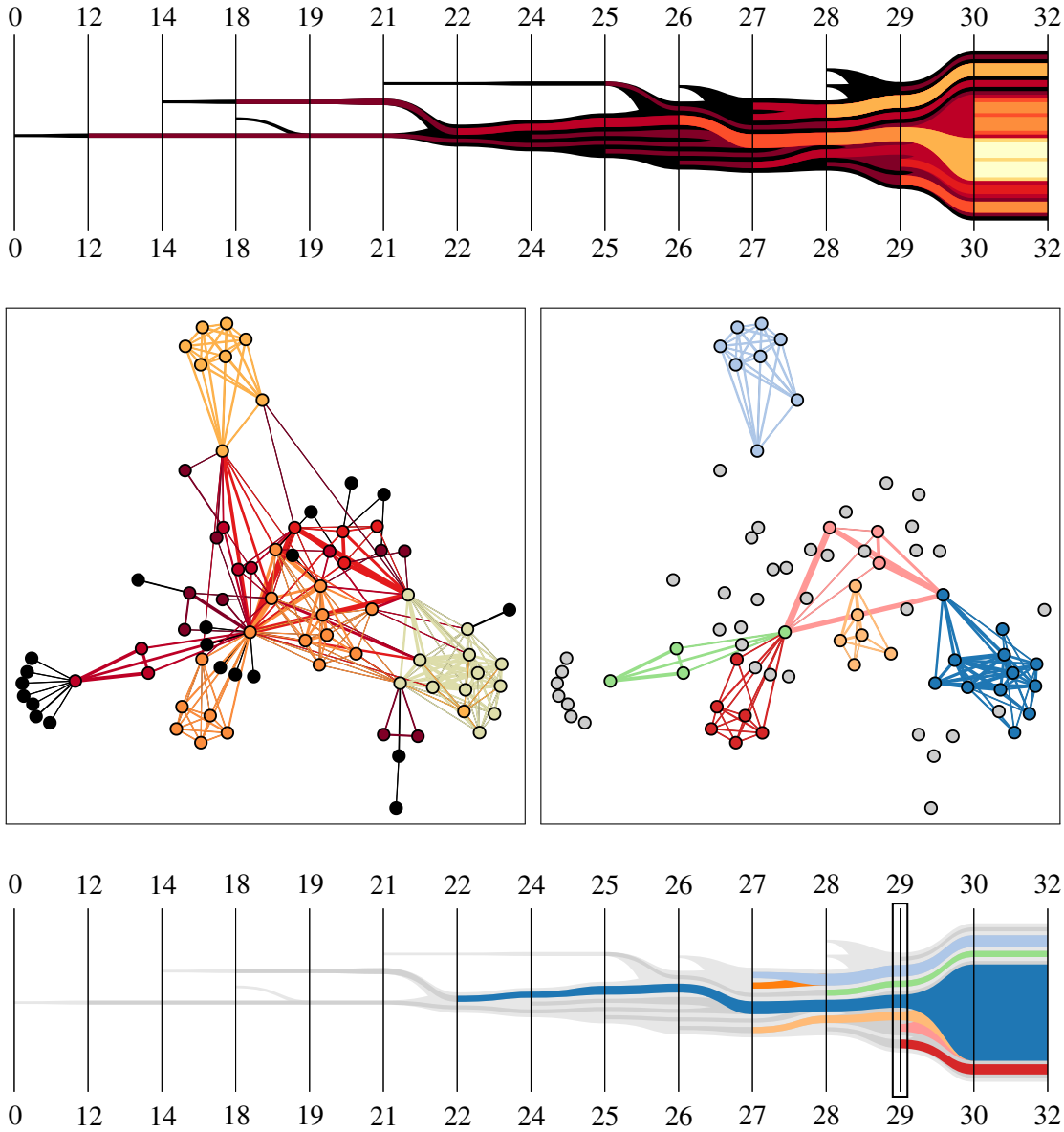


Figure 3.7: Illustrations of the evolution of clique communities in the "Les Misérables" co-occurrence network. The images in the center render the network with a force-directed layout, where the left image colors simplicies based on the largest community degree they participate in, and the right image uses colors to distinguish between individual communities for edge weight threshold 29 and degree 4. The NTGs comprehensibly illustrate when and which clique communities merge while increasing the edge weight threshold (left to right), as well as how communities for different degrees are nested inside each other (layers). Note, communities are allowed to share simplicies as individuals interact with various groups of distinct interconnectivity. Users can explore the dataset by interacting with the NTG. For example, the bottom NTG highlights individual 4-clique communities for threshold 29 that are also displayed in the right force-directed layout.

Integrating the graph visualizations into a visual analytic interface enables users to explore different edge weights and community degrees. Intuitively, varying the edge weight threshold helps to extract the "core" of a community, while changing the degree permits analyzing the same social circle according to different granularity levels by revealing the sub-communities it consists of. For instance, analyzing the network for the largest edge weight results in a clear community structure for $k = 4$ (Fig. 3.7 center right, and bottom). By leaving k fixed and moving "horizontally" in the nested graph, it is possible to track the evolution of a selected community. E.g., by moving from threshold 32 to 29, the big dark blue community at threshold 32 actually consists of three different sub-communities (blue, light orange, and light red). Moreover, the six communities that exist for threshold 28 turn out to be highly relevant for the structure of this network, as each of them corresponds to a significant group of characters: the members of the revolutionary association called *Les Amis de l'ABC* (dark blue), the circle of friends of the young *Fantine* (light blue), the members of the *Patron-Minette* crime gang (orange), the social circle of Bishop *Myriel* (green), the participants at *Champmathieu*'s trial (red), and the family of *Marius* (light red). In conclusion, NTGs comprehensibly illustrate the evolution of communities for varying edge weight thresholds and degrees simultaneously, and can be used as interactive steering devices for visual analysis.

3.4 DISCUSSION

This chapter presented nested tracking graphs (NTGs): a novel topological abstraction that records and visualizes the evolution of features that exhibit a nesting hierarchy, e.g., sublevel and superlevel set components for multiple levels. Thus, NTGs set multiple tracking graphs in context to each other, and provide a compact visualization that enables users to follow the evolution of features and their properties for different levels simultaneously. It was demonstrated that NTGs can be used in various applications as an effective tool for interactive data exploration and analysis.

The initial NTG visualization algorithm presented in Sec. 3.2.3 computes smooth and steamlined layouts, but they still contain numerous edge crossings. For example, the red lines of Figure Fig. 3.6 between timestep 835 and 840 cross even though their corresponding components do not merge. These "false" crossings—which could be misinterpreted as merge or split events—can only occur between timesteps and never at an exact timestep. Hence, edge crossings between timesteps are only layout based and do not have any semantic interpretation. Some edge crossings are unavoidable, but many of them result from the individual layout computation of the levels. To reduce the number of edge crossings it is necessary to consider the graph structures of other levels

while computing the final layout. Based on this idea, Köpp et al. [48] recently proposed an improved NTG layout algorithm that uses a custom optimization procedure based on simulated annealing [47]. To improve the layout further, in the future edges could be arranged according to the spatial position of their associated features. To cope with a large number of timesteps and components it also seems possible to apply edge bundling techniques that summarize time intervals and branches of the graphs. Edges and nodes of the graph could also be linked to other visualizations such as persistence diagrams and histograms to provide additional information about the components.

Another limitation is the maximum number of visible levels, which depends on the dataset, the amount of cluttering of the resulting graphs, and on whether the graphs are shown statically or in an interactive interface. Based on the presented experiments, static drawings of nested graphs should not show more than three levels at once. An interactive interface can use zooming and focus-and-context techniques to compensate for cluttering, making it practical to show up to 8 levels in a nested graph. However, using only two levels is already a significant improvement over previous visualization techniques as the nested tracking graph shows the tracking graphs for both levels simultaneously and sets them in context to each other.

The prime application of NTGs is to characterize the evolution of sublevel and superlevel set components for multiple levels. Obviously, the choice of these levels has a significant impact on the resulting visualization, and updating the levels requires recomputing the tracking information across all timesteps and levels. The approach presented in Ch. 5 improves on these limitations by determining significant levels based on the merge tree structure of the underlying scalar field, and by utilizing an intermediate data structure that enables the fast computation of NTGs. As demonstrated in Sec. 3.3.4, NTGs can be applied in various other fields as well. In general, they are suited to visualize any time-varying hierarchies; such as the ones present in hierarchical clustering, hierarchical diffusion, and threshold based methods.

CHAPTER 4

VOIDGA: A VIEW-APPROXIMATION ORIENTED IMAGE DATABASE GENERATION APPROACH

This chapter presents a novel view-approximation oriented image database generation approach (VOIDGA) that enables the adequate generation of arbitrary view angles [64]. The approach utilizes Depth Image Based Rendering (DIBR) techniques to derive novel views based on a set of depth images (Sec. 2.6). In contrast to approaches that store a huge amount of images to cover a wide range of possible view directions (Sec. 4.1), VOIDGA identifies and stores a reduced set of images that enables the approximation of any view angle with an acceptable visual error (Sec. 4.2). This further reduces the size of image databases and the number of images that need to be processed by DIBR algorithms. VOIDGA is demonstrated on several challenging real-world examples (Sec. 4.3), and the resulting view approximation quality is examined qualitatively and quantitatively via two image-based similarity metrics (Sec. 4.4).

4.1 MOTIVATION

The increasing size and complexity of datasets make it necessary to reduce the amount of stored information while still supporting effective data exploration through interactive visual interfaces. Especially in the case of large-scale simulations, it is often impossible to store entire simulation states for *post hoc* analysis due to bandwidth and I/O constraints. To address this issue, Ahrens et al. [2] proposed the ParaView Cinema concept as an image-based approach for the *post hoc* exploration of simulation output (Sec. 2.5). In their approach, an image database is created *in situ* consisting of color and depth images of simulation elements taken from various camera positions. Such databases are several orders of magnitude smaller than the simulation data they are derived from, and they enable the real-time exploration of extreme scale simulations by querying and compositing images from the database. Rudimentary image database viewers facilitate basic camera movement by simply snapping to the closest available camera position for a requested viewpoint [3, 63, 82]. As a principled limitation, these viewers cannot visualize viewpoints that had not been foreseen and specified during database generation. However, depth image based rendering (DIBR) algorithms (Sec. 2.6) enable the approximation of novel views based on existing database elements, which supports unconstrained camera interaction for visual exploration. In turn, such algorithms introduce approximation errors that depend on the quality of the used DIBR technique and the input depth images. It is also not clear which and how many images are needed to adequately approximate a wide range of novel views.

The proposed approach addresses these issues by taking the first step towards leveraging the information stored in an image database to its full potential. Specifically, this chapter describes a novel view-approximation oriented image database generation approach (VOIDGA) that determines a minimal set of input depth images that enable the approximation of new views within a certain error bound. The core concept of VOIDGA is to identify and store images that significantly contribute to the overall view-approximation quality, while at the same time discarding images that can already be adequately approximated. This yields much smaller image databases than the ones produced by current state-of-the-art implementations which uniformly sample images on a spherical grid. This also results in a reduced set of images that need to be processed by DIBR algorithms while still guaranteeing a minimum approximation quality.

4.2 APPROACH

Alg. 7 outlines the novel view approximation-oriented image database generation approach (VOIDGA) that utilizes depth image based rendering (DIBR) methods (Sec. 2.6) and image similarity metrics (Sec. 4.2.1) to identify a reduced set of depth images $\hat{\mathcal{D}}$ that enable the adequate geometry approximation of the depicted simplicial complex $\mathcal{K}$. VOIDGA is essentially a Greedy algorithm that iteratively refines a sampling grid $\mathcal{G}$ and then only stores images that significantly contribute to the overall *post hoc* approximation quality. To this end, VOIDGA consists of three phases: the database backbone generation (Sec. 4.2.2; lines 1-2), the database refinement (Sec. 4.2.3; lines 3-17)), and the database downsampling (Sec. 4.2.4; line 14). To run VOIDGA completely automatically, it is necessary to specify the maximum number depth images N, the initial (and thus maximum) image resolution R, and the error thresholds of the used image similarity metrics E. In the following, VOIDGA is demonstrated with the DIBR techniques described in Sec. 2.6—which features splatting (Fig. 2.26, top right) and depth image triangulation (Fig. 2.26, bottom)—as well as the Multi-Scale Structural Similarity Metric and Average Depth Difference. However, it would also be possible to use any DIBR method or image metric due to the modular design of VOIDGA.

Algorithm 7: VOIDGA(Grid $\mathcal{G}$, Simplicial Complex $\mathcal{K}$, Thresholds (N, R, E))

```
1  // Backbone Generation
2  D̂ ← RenderGroundTruth( K, G, R )
3  // Refinement
4  do
5      // Get Candidates
6      G ← RefineGrid( G )
7      D ← RenderGroundTruth( K, G, R )
8      TuneApproximationRenderer( D̂, D, G, R )
9      while |D| > 0 do
10         D̂′ ← RenderApproximation( D̂, G, R )
11         (D, Ê) ← GetWorstApproximatedGroundTruthImage( D̂′, D )
12         // If error or size threshold reached return downsampled images
13         if Ê < E ∨ |D̂| > N then
14             return ReduceImageResolutions( D̂, E_M, E_A )
15         // Update depth image sets
16         D ← D \ {D}
17         D̂ ← D̂ ∪ {D}
```

4.2.1 Image Similarity Metrics

To assess the significance of individual images during database generation, VOIDGA compares the current view approximations against ground truth renderings via some image similarity metrics. Suitable metrics that are used by default are the Multi-Scale Structural Similarity Metric (MS-SSIM) [111] that estimates the perceived image similarity, and the Depth Difference (DD) [49] that measures the actual shape distortion.

The DD tries to capture the volumetric difference between two objects by comparing multiple depth images of the objects in pairs, where the images of each pair are generated with the same camera calibration, i.e., location, up-vector, direction vector, near-far plane, and so forth. Similar to Cinema databases, these pairs are generated on vertices of a grid that encapsulates the datasets and aim towards the object center (Fig. 2.23). The difference between two depth images is then measured by the so-called Average Depth Difference (ADD) that is the sum of the actual depth value difference per pixel. Hence, it is assumed that the depth images have the same size, and that their values are in the range $[0, 1]$. The DD is then given as the average of all computed ADDs, where a value of 0 implies that the depicted objects are identical, and larger values are proportional to shape variations.

The MS-SSIM is modeled after the assumption that the human visual system is highly adapted for extracting structural information from 3D projections. Thus, a measure of the structural similarity between images can provide a good estimate of the perceived image quality [110, 111]. In contrast to the original SSIM [110], the MS-SSIM iteratively downsamples the input images to determine the luminance and contrast variations for varying resolutions. This allows to evaluate the structural similarity between images more independently from the actual image sizes. Similar to the ADD metric, VOIDGA computes the MS-SSIM for multiple camera positions and builds the average to evaluate the structural similarity across the entire approximation. However, the ADD computes an error value from 0 (identical) to 1 (complete opposite), whereas the MS-SSIM computes a score from 0 (not similar) to 1 (identical).

4.2.2 Database Backbone Generation

The first stage of VOIDGA normalizes the dataset geometry according to the dimensions of the unit-cube with center at the origin, and then select a sampling grid structure. A common way to generate Cinema databases is to uniformly sample along a latitude-longitude parameterized sphere that encapsulates the dataset, where the cameras are positioned at the grid vertices and aim towards the center (Fig. 2.23). For image database viewers that

simply snap to the next available image, this creates intuitive transitions as it seems like the camera rotates along the lat-lon axes. However, this grid causes an oversampling at the poles, and an undersampling at the equator (Fig. 4.1 left). Since DIBR methods are not restricted to the actual image locations, it makes more sense to use an icosahedron as a sampling primitive. In contrast to a lat-lon grid, each icosahedron refinement uniformly creates new sampling positions that equally cover possible view angles (Fig. 4.1 right). As these positions are eventually ADDed to the database, VOIDGA effectively improves the approximation quality in each step.

To generate the database backbone (a small set of images that are the basis for the view approximation), VOIDGA samples images on the 12 vertices of the unrefined icosahedron (intersection of red lines in Fig. 4.1 right). Then, a depth image with the initial resolution is generated for each vertex of the grid using an orthographic camera, where the camera width and height are set to the icosahedron diameter. Thus, each image encapsulates the complete dataset, and the 12 locations already provide a good view angle coverage. It is also possible to ADD model-specific view angles—such as interior locations—to the backbone if such important angles are known a priori. The next phase uses the resulting images as a basis for the view approximation.

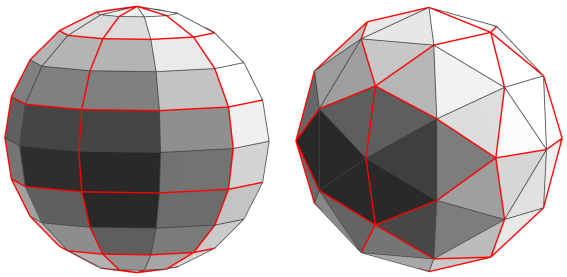


Figure 4.1: Sampling grids for the Cinema database generation: a latitude-longitude parameterized sphere (left), and a refined icosahedron (right). The first subdivision is shown in red, and the second subdivision in gray. The icosahedron vertices are uniformly spread, while the latitude-longitude grid oversamples poles, and undersamples the equator.

4.2.3 Database Refinement

In this phase, VOIDGA iteratively refines the sampling grid $\mathcal{G}$ (line 6) and only ADDs images to the database that significantly contribute to the global approximation quality. Hence, it is necessary to derive two depth images per sampling location: the depth images $\mathcal{D}$ of the ground truth geometry $\mathcal{K}$ (line 7), and the depth images $\hat{\mathcal{D}}'$ of the current view approximation using all available images in the database $\hat{\mathcal{D}}$ (line 10). Instead of storing all new depth images $\mathcal{D}$ immediately in the database, VOIDGA iteratively determines the worst approximated ground depth image and ADDs it to the database. Note, the approximated depth images $\hat{\mathcal{D}}'$ depend on the used DIBR algorithm (e.g., triangulation

or splatting) and its respective parameters (e.g., the distance threshold and point size). To automatically determine suitable DIBR parameters, VOIDGA iteratively tunes the DIBR parameters until it finds a local approximation error minimum (line 8). To this end, each iteration compares the ground truth images to the current view approximation results by computing the ADD and the MS-SSIM for each resulting pair. As soon as the error increases, the automatic tuning is stopped, and the current error and DIBR settings are communicated to the user. Although VOIDGA can run fully automatically, this gives users the option to directly compare the current approximation against the ground truth every time the grid is refined; either by directly contrasting the pairs, or by free camera movement as long as the ground truth can be rendered at interactive framerates. Moreover, users can adjust the database constraints and the error thresholds, which is especially useful if proper initial settings are unknown.

After the automatic tuning, VOIDGA renders the approximated depth images $\hat{\mathcal{D}}'$ at the new sampling locations (line 10), and then selects the ground truth image D that currently exhibits the largest approximation error $\hat{E}$. If the current worst-case approximation error $\hat{E}$ is below the threshold E, or if the database size exceeds the image limit N, then VOIDGA advances to the final stage. Otherwise, D is removed from the list of candidates $\mathcal{D}$ and is ADDed to the database $\hat{\mathcal{D}}$. This process then repeats until either the thresholds are satisfied, or all candidate are ADDed. In the latter case, the sampling grid is refined once more and the process repeats.

4.2.4 Database Downsampling

Finally, VOIDGA communicates to the user the impact of the image resolutions on the overall approximation quality and the used disk space. Obviously, a lower image resolution results in worse approximations, but the benefit of a significant disk space gain might be worth a slightly worse approximation quality. Note, for this stage it is not necessary to actually recompute the depth images D and $\hat{D}$, instead they can be directly downsampled from the high-res images.

Overall, this approach reduces the number of stored images, while asserting a minimal approximation quality at the missing sampling locations. In the experiments presented in Sec. 4.3, sequentially executing this process took roughly one minute. Note, however, deriving new depth images and their scores is embarrassingly parallel, and therefore VOIDGA is much more efficient in practice.

4.3 RESULTS

In the following, the effectiveness of VOIDGA is demonstrated and evaluated on several real-world examples of varying complexity; including smooth surfaces (Sections 4.3.2 and 4.3.3), jagged surfaces (Sec. 4.3.4), and sparse line geometry (Sec. 4.3.5).

4.3.1 Error Plots

To evaluate the approximation quality of the following experiments, the resulting error is examined qualitatively and quantitatively based on the two image similarity metrics that have already been introduced in Sec. 4.2.1, i.e., the Multi-Scale Structural Similarity Metric (MS-SSIM) and the Average Depth Difference (ADD). Specifically, for a given image database that was either generated by uniform samples (U) or via VOIDGA (V), a total number of $1,000$ random view directions are derived on the surface of the unit sphere, which are then used to generate ground truth renderings and approximated views. Subsequently, the resulting images are compared with the image similarity metrics, which yields a histogram for each database and its parameters. Figures 4.2 and 4.3 depicts the results for a variety of databases, which are first grouped metric (individual figures), then by dataset (rows), and then by approximation method (columns). Each of these database groups are further subdivided by the used image resolution (left to right), and the sampling method (colors).

The individual plots are discussed in their respective dataset section. Note, however, that the databases generated by VOIDGA are always biased towards the approximation method and error metric that was used during database generation, i.e, towards triangulation and MS-SSIM for the viscous finger and ground water dataset, and towards splatting and ADD for the jet dataset. Moreover, VOIDGA can obviously not outperform the maximum refinement as it only collects a subset of these images. Yet, VOIDGA performs in all case studies almost as good as the maximum refinement, although it uses only around halve as much images.

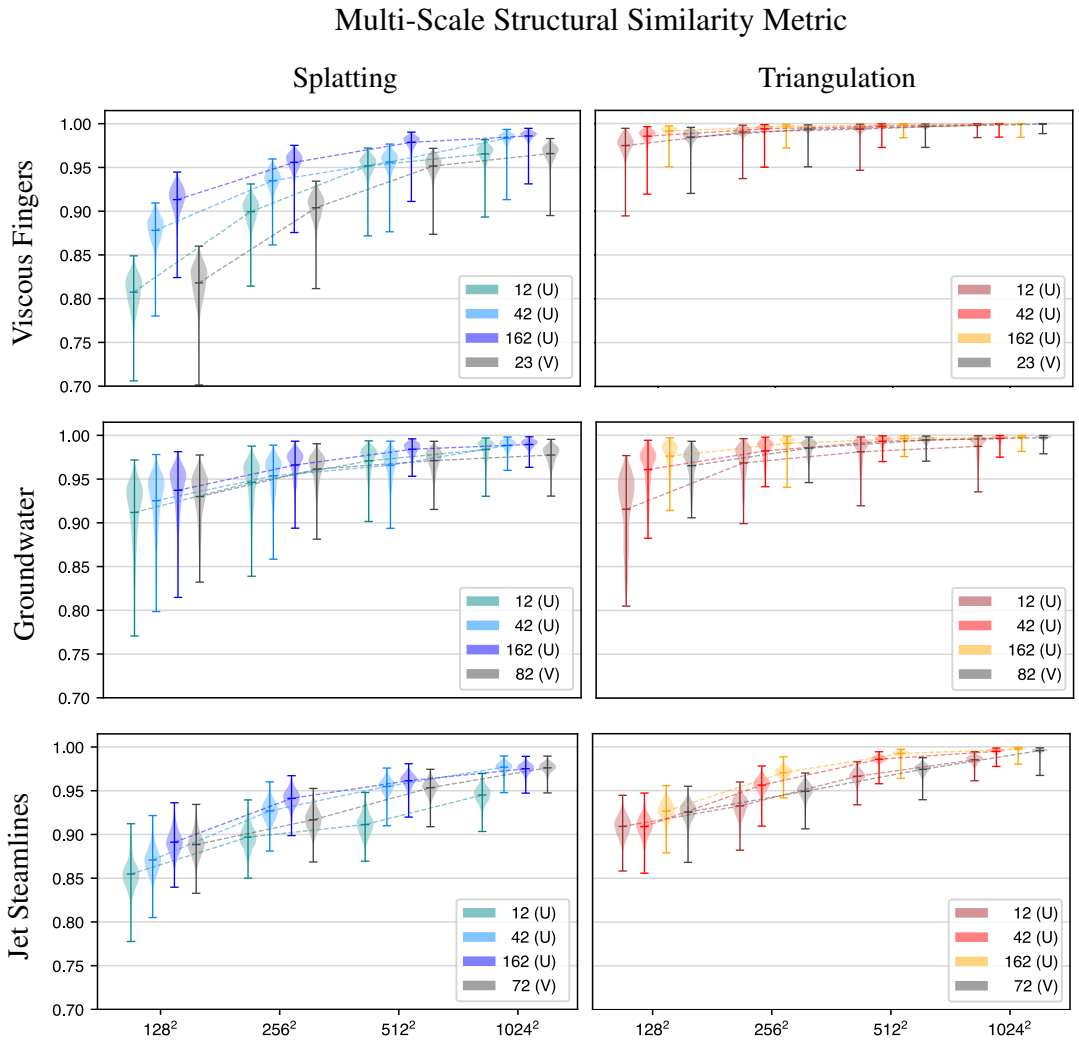


Figure 4.2: Approximation errors of the resulting image databases measured via the Multi-Scale Structural Similarity Metric (MS-SSIM), and the Average Depth Difference (ADD; Figure 4.3) for 1,000 random viewing positions on the unit-sphere. For each metric, the error is computed for the splatted (cool colors, left column) and the triangulated approximation (warm colors, right column) grouped first by dataset (rows), then by image resolution (x-axis), and finally by the selection method of database elements (individual bars). Legends indicate how many images were used as the basis for the view approximation. The violin plots illustrate for each case the histogram of errors over the random positions. This figure shows the MS-SSIM metric (higher values are better; 1.0 denoting identical images). It can be observed that for an increase in image and sampling resolution the approximations converge to the ground truth, where VOIDGA databases are biased towards the approximation method and the error metric that was used during the optimization process, i.e., towards triangulation and MS-SSIM for the viscous finger and ground water dataset, and towards splatting and ADD for the jet dataset.

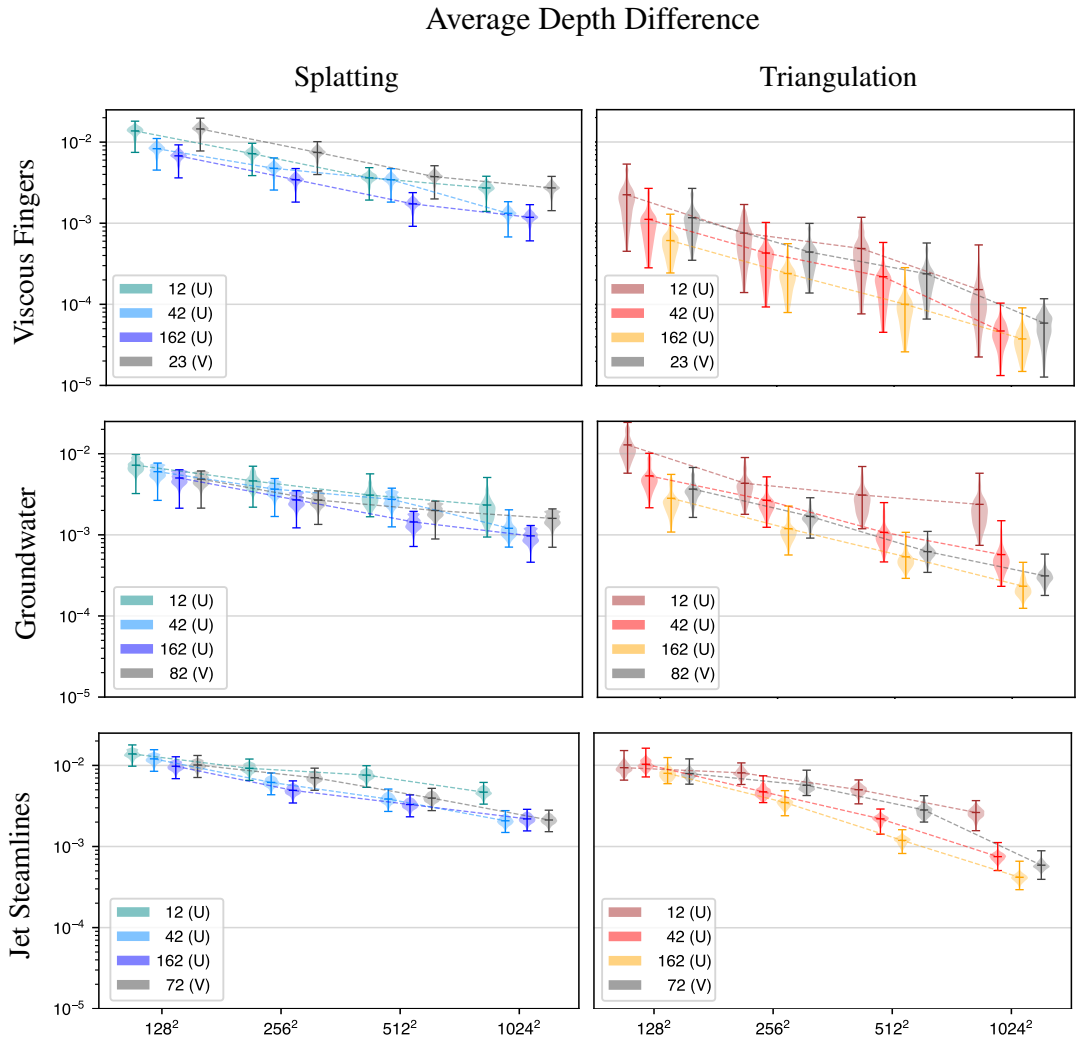


Figure 4.3: ADD metric in logarithmic scale (lower is better, 0.0 denoting perfect reproduction). Again, VOIDGA is biased towards the used approximation method and error metric during the optimization process, and therefore performs close to the maximal sampling in these categories. As VOIDGA selects images in a Greedy approach, the algorithm might select more images than strictly necessary, but can be employed *in situ* without prior knowledge of the values underlying these diagrams. Moreover, some specific view angles might not have a huge impact on the total approximation quality—e.g., cameras that look into a cavity of the ground water dataset—but are still ADDed to the database by VOIDGA. Except for the streamline dataset, the top-heavy histograms for both metrics indicate that the distribution is skewed strongly towards higher similarity with only few outliers. It becomes apparent that even without employing VOIDGA, databases consisting of only 42 depth images with a resolution of 256^2 pixels yield adequate results.

4.3.2 Viscous Fingering

The first case study demonstrates VOIDGA in an ideal application scenario: datasets that exhibit large smooth surface areas. Specifically, VOIDGA was used to generate a minimal image database for the viscous finger simulation ensemble that was already introduced in Sections 2.4.2 and 3.3.1. Utilizing again the approach of Lukasczyk et al. [63], viscous fingers can be identified as superlevel set components of the salt concentration density estimates. Fig. 4.4 shows the ground truth isosurface geometry and the view approximations, where the viscous fingers and the salt supply are colored bright orange and dark gray, respectively. Quantitative results for this dataset are shown in the left column of Figures 4.2 and 4.3. For smooth surfaces as the ones found in this case study, triangulation outperforms the splatting technique. Not only does it exhibit a lower approximation error, but also achieves a higher frame rate. Splatting causes a warp of the original surface—i.e., creates an artificial width of the surfaces based on the point size—which causes the generated views to score lower on the image metrics. As shown in Figures 4.2 and 4.3, VOIDGA uses fewer images (23) than the complete second icosahedron refinement (42), yet achieves similar error scores. Images that depict the top of the salt supply are discarded by VOIDGA as the smooth surface of the supply can already be approximated by the database backbone. The ADDitional images correspond to view angles that were approximated badly before, but do not have a significant impact on the overall approximation quality.

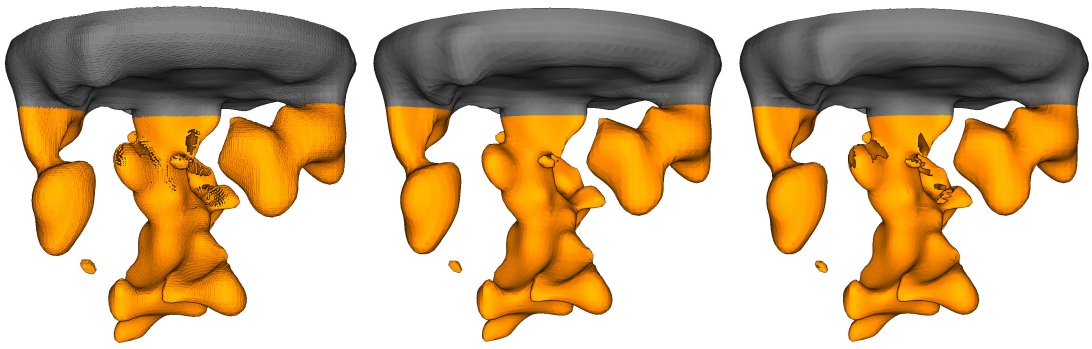
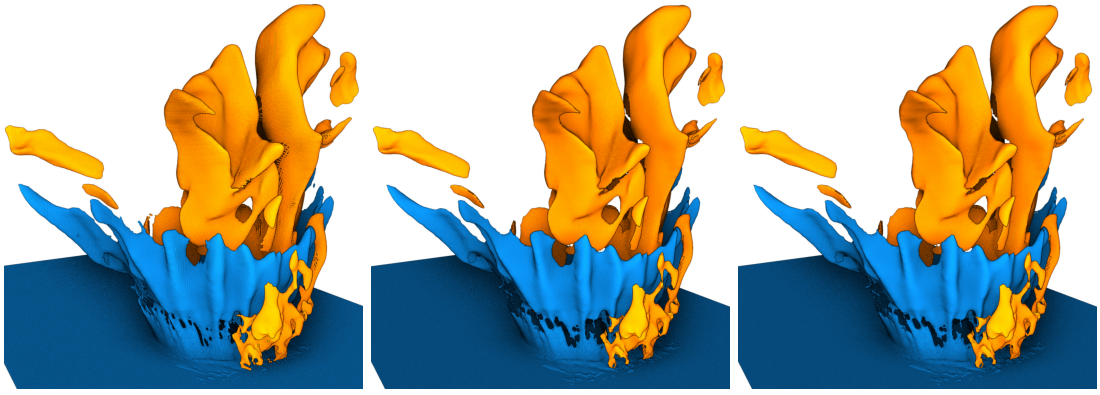


Figure 4.4: Comparison between the generated views (left and right) and the ground truth (middle) for a random time step of the viscous finger dataset, where fingers are shown in orange, and the salt supply in gray. The views are approximated using only 23 depth images with a resolution of 256^2 pixels that were chosen by VOIDGA.

4.3.3 Asteroid Ocean Impacts

A second dataset exhibiting large and relatively smooth contours is part of a threat assessment study of asteroid ocean impacts [85] that was made publicly available for the 2018 scientific visualization contest [43]. The dataset consists of several extreme scale simulations that model different impact scenarios for varying impact angles, asteroid sizes, and heights of potential airbursts. Fig. 4.5 depicts contours for the temperature and water density field for impact scenario *yA*31, i.e., no airburst event, an asteroid diameter of 250 meters, and an entry angle of 45 degrees. Specifically, the temperature and water density contours for level 0.2 eV and 0.002 g/cm^3 are shown in orange and blue, respectively. The contours consist of roughly three million triangles that use up around 110 MB uncompressed space, while the generated view was derived by only 12 depth images with a resolution of 512^2 pixels for each contour, i.e., 24 depth images with an uncompressed total size of 24 MB. The images were chosen using VOIDGA to ensure approximation error bounds of 0.001 ADD and 0.97 MS-SSIM for the current view. Large surfaces are accurately approximated, while the base of the water vapor exhibits some approximation errors. Triangulations do not create adequate surface patches of small features due to the low pixel density of the used depth images. Splatting, on the other hand, renders points at the location of small features as long as they are depicted by a depth image pixel. However, the splatted surface contains gaps that need to be filled by increasing the point sizes, which in turn causes an artificial surface warp.



Splatting Ground Truth Triangulation

Figure 4.5: Comparison between the generated views (left and right) and the ground truth (middle) for the asteroid impact dataset *yA*31 at cycle time 29945. The orange and blue surfaces are contours of the temperature (0.2 eV) and water density field (0.002 g/cm^3), respectively. Views have been approximated using only 24 depth images with a resolution of 512^2 pixels that have been chosen by VOIDGA to bound the maximum approximation error for the current view.

4.3.4 Karst Limestone Ground Sample

This case study demonstrates that the proposed approach can also approximate very complex surfaces with an acceptable error, and that it enables the composition of approximated and explicitly stored geometries. To this end, a Cinema database was created for a karst limestone ground sample that was taken in south Florida. The ground sample was provided by the Texas Advanced Computing Center (TACC) and the Florida International University as a triangulated surface consisting of roughly 8 million triangles (gray surface of Fig. 4.6 top). Domain experts involved in this research are primarily interested in the propagation of ground water through the stone cavities (red streamlines of Fig. 4.6). This dataset is challenging for depth image based geometry approximation since the complex structure of the cavities occlude most of the interior geometry. To compensate, it is usually necessary to sample depth images on a dense grid. VOIDGA, on the other hand, determines a small set of samples that adequately reconstruct the outer shell of the stone. However, to also demonstrate the effects of undersampling, this case study uses only 42 depth images with a resolution of either 512^2 or 128^2 pixels that are sampled on a once subdivided icosahedron (Fig. 4.1b).

The middle and bottom of Figure Fig. 4.6 show approximated views based on depth image triangulation and splatting, respectively. The lighting of the complete scene is performed in the post processing shader where screen space ambient occlusion greatly enhances the perception of the stone porosity and the spatial arrangement of the streamlines. The uniform camera samples accurately reconstruct the outer structure of the stone, but they do not depict the interior geometry of most cavities. Moreover, fine details of the structure are only visible if the resolution of the depth images is high enough. Since the proposed triangulation algorithm requires at least three neighboring depth pixels that are below the distance threshold to create a surface patch, the resulting approximations ignore one pixel wide surface depictions (Fig. 4.6 middle right). Splatting preserves these features, as each depth image pixel is still represented by a single point (Fig. 4.6 bottom right). However, the size of these points must be large enough to fill the gaps between points, which gives the incorrect impression that surfaces have a thickness. Yet, this greatly improves the 3D perception and emphasizes hard edges such as cavity borders. Nevertheless, rendering the point cloud is more expensive than rendering the triangulation. The triangulation also enables the more accurate estimation of surface normals since splatting renders each point as a flat disc that faces the camera, which causes depth discontinuities at the disc boundaries, and thus a rough surface appearance.

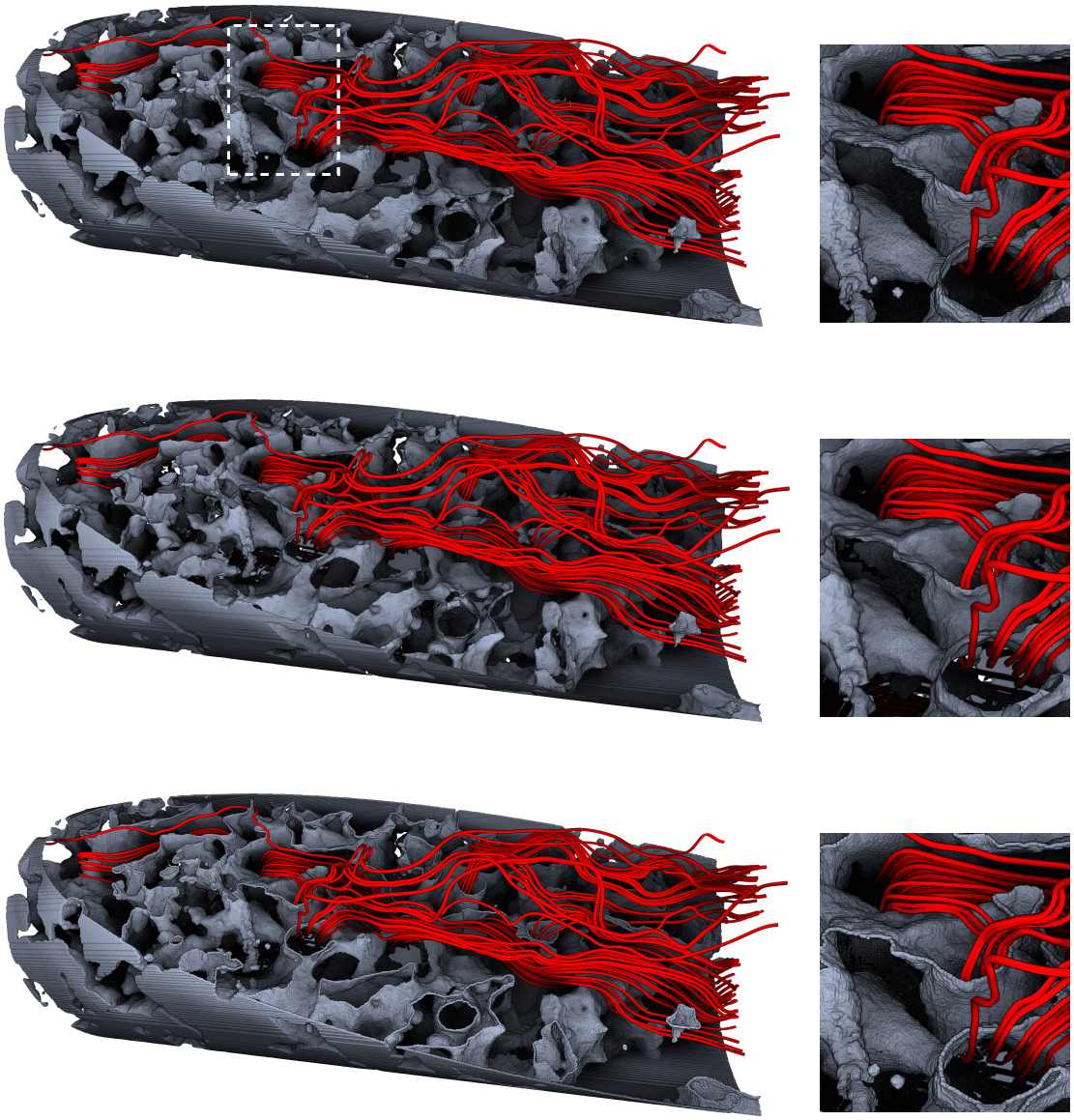


Figure 4.6: View approximations of the limestone dataset (top) based on 42 depth images with a resolution of 512^2 pixels that are sampled on the vertices of a once refined icosahedron. The dataset was provided by the Texas Advanced Computing Center (TACC) and the Florida International University. The images show the path of water (red streamlines) through a karst limestone ground sample (gray) that was taken in south Florida. As the approximations show a view angle that was not covered by the used depth images, the approximation error is largest in the cavities where no geometric information is available. Nevertheless, the outer structure is accurately reconstructed even for the relatively low number and resolution of the depth images. Deploying VOIDGA improves the approximation quality as it ADDs only view angles that contribute the most to the current approximation, i.e., view angles that look into currently undepicted cavities.

To emphasize the impact of the image resolution, Fig. 4.7 shows the geometry approximations that result from using images with only 128^2 pixels. These depth images do not have a sufficient resolution to represent small cavities, as neighboring pixels are too far apart in world space, while the drastically varying surface between the pixels is not depicted. However, even at this resolution, prominent features such as the big cavities are clearly identifiable. Image triangulations requires a fairly high distance threshold to coincide with the rough shape of the original surface, which results in numerous distorted triangles. Splatting also requires a large point size to fill gaps. Because the triangulation connects neighboring pixels with similar depth values, splatting produces much better results for low resolution images as each pixel is still mapped to 3D space independent of its neighbors at the price of warping the resulting surfaces. Therefore, the splatted representation coincides better with the shape of the original triangulation, but the large points let the surfaces appear very thick. This is also reflected in the error metrics (Figures 4.2 and 4.3, second row). Especially the MS-SSIM is very sensitive to the surface warps. Furthermore, a database derived by VOIDGA uses far fewer images (82) than the maximum refinement level (162) while achieving similar approximation errors. Since the cylindrical stone sample has a relatively smooth backside, VOIDGA primarily stores images that depict the front and the inside of cavities.

An advantage of the modular design of the demonstrated DIBR algorithm is that the approximated geometry can be rendered together with non-approximated geometry. For instance, the red streamlines of Fig. 4.6 are explicitly stored geometries that are correctly composed with the approximated geometry. Based on this principle, extremely large simulation elements can be approximated by depth images, while specific features of smaller size can be stored explicitly.

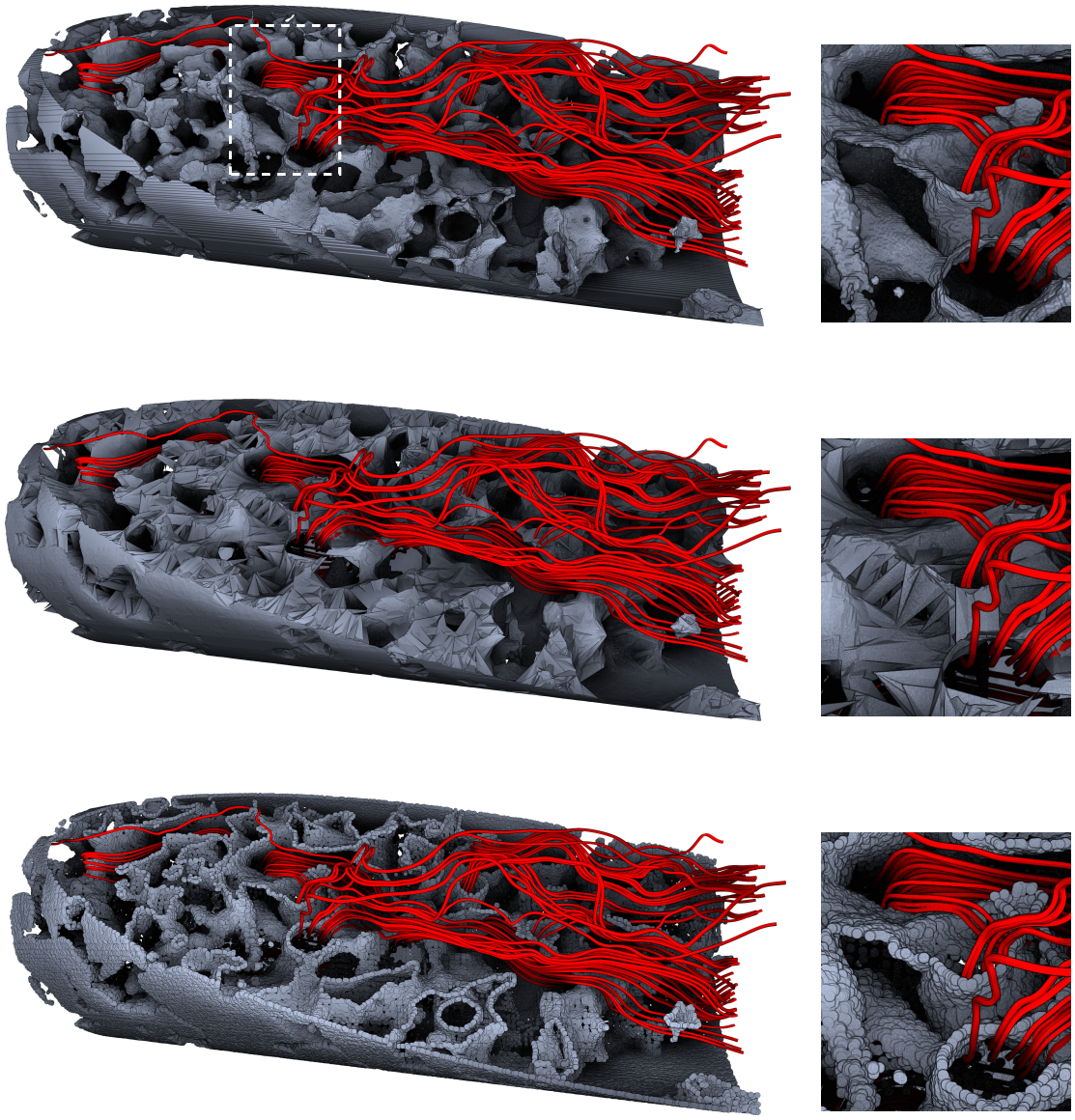


Figure 4.7: Illustration of view approximations generated by depth images with an insufficient image resolution. In this case, the views have been generated with 42 depth images with a resolution of only 128^2 pixels. For such low resolutions, splatting (bottom) produces far better results since points are rendered independently at the locations of every depth image pixel, while image triangulations (middle) require at least three neighboring pixels that span a non distorted triangle. However, the huge point sizes have a drastic impact on the used error metrics.

4.3.5 Jet Streamlines

Sparse line geometry is another challenge for DIBR due to the strong depth variations of neighboring pixels. Therefore, this case study examines a set of streamlines that correspond to particle paths of a CFD *Jet* simulation (Sec. 3.3.2).

For this case study, VOIDGA was tuned to generate a database with a focus on high quality depth approximations rather than image similarity, effectively de-emphasizing color reproduction. This can be done by enforcing a strict ADD threshold (0.0004) and a relaxed MS-SSIM threshold (0.8). Still, the approximations cause large errors for small image resolutions (Figures 4.2 and 4.3, right column) as they are not sufficiently large enough to distinguish between individual streamlines. Thus, the image triangulation falsely connects neighboring streamlines via surface patches (Fig. 4.8 right). Although splatting can still produce convincing results, the necessary large point size bloats the streamlines (Fig. 4.8 left), which has a significant impact on the error metrics. Yet, the VOIDGA database (72 images) and the maximum refinement level (162 images) achieve similar errors. The colors of the streamlines encode their lifetime and are mapped *post hoc*. This requires to store for each depth image an ADDitional floating point image that records at each pixel the lifetime of the depicted part of the streamlines. The ability to apply a color map *post hoc* on the approximated geometries demonstrates that the proposed approach can be easily combined with the existing practice of image databases.

Splatting Ground Truth Triangulation

Figure 4.8: Comparison between the generated views (left and right) and the ground truth (middle) for the jet dataset. To emphasize potential visual errors, the views have been approximated by using only the database backbone (12 depth images) with a resolution of either 256^2 or 512^2 pixels for the splatting or triangulation technique, respectively.

4.4 DISCUSSION

This chapter presented a novel view-approximation oriented image database generation approach (VOIDGA) that determines and stores a minimal set of images for the generation of arbitrary views while bounding the maximum approximation error. As demonstrated on several challenging real-world examples, VOIDGA can reduce image database sizes and the number of images that need to be processed by DIBR methods. VOIDGA can also ensure that a disk space budget is used to its full potential, which stands to be useful for *in situ* visualization, but also for sharing visualization results where bandwidth usage is of importance. The resulting approximation errors were examined qualitatively and quantitatively via two image-based comparison metrics: the ADD and MS-SSIM. The results indicate that a relatively low number of database elements (~ 42) at medium resolution ($\sim 512^2$) can already produce high quality approximations. Moreover, smooth surfaces can be well approximated by triangulations, whereas extremely jagged surfaces, sparse line-geometry, and low-resolution depth images are best approximated by splatting.

Towards adapting VOIDGA for production use, many improvements appear possible. Due to the modular design of VOIDGA, it is possible to integrate more advanced DIBR methods and other error metrics to further improve the resulting approximation quality. VOIDGA could also store view-dependent resolutions and feature-based camera locations. For example, depictions of smooth surfaces could be stored at low resolution, whereas detailed surface variations and important features are depicted by high-res images. However, as VOIDGA builds on top of DIBR techniques, it is necessary to derive and store depth images. Thus, VOIDGA does currently not support the approximation of volume renderings and transparent geometry. In this case, one needs to adapt other techniques such as image warping [50] or volumetric depth images [9, 34].

VOIDGA was demonstrated with fairly rudimentary DIBR approaches (depth image triangulation and splatting). Although they require a minimal overhead and already produce acceptable results, more advanced DIBR methods are expected to produce higher quality approximations. Such techniques can easily be integrated into VOIDGA due to its modular design. Moreover, both presented DIBR implementations require parameters (the distance threshold and the point size) that have a significant impact on the resulting approximation quality. VOIDGA is capable of automatically finding suitable initial parameters, but the current tuning procedure can get stuck in local extrema. To solve this problem, it is necessary to deploy more advanced optimization techniques such as simulated annealing [47].

Naturally, the effectiveness of VOIDGA depends strongly on the used comparison metrics. Although the demonstrated image metrics measure geometry representation (ADD) and image similarity (MS-SSIM), both are not without drawbacks. The most significant problem is their strong dependence on the background to foreground ratio. In effect, a larger background will result in better similarity scores which is not ideal for real-world settings. Moreover, the ADD is computed for normalized depth values, and therefore depends on the precision of the depth buffer. The MS-SSIM, on the other hand, requires input parameters that can currently only be chosen heuristically [111]. Furthermore, both metrics evaluate the overall image quality, and thus neglect small but potentially important features. Therefore, it is necessary to develop and integrate other image metrics to improve the optimization process.

CHAPTER 5

DYNAMIC NESTED TRACKING GRAPHS

This chapter combines Nested Tracking Graphs (Ch. 3) and Cinema Databases (Ch. 4) to enable the interactive *post hoc* visual analysis of large-scale simulations with numerous superlevel set components (Sec. 5.1). The approach first derives, at simulation runtime, a specialized Cinema database that consists of various rendering and analysis products, including images of component groups, merge trees, and intermediate data structures that store tracking information (Sec. 5.2). This database is processed *post hoc* by an efficient graph operation-based algorithm to dynamically compute nested tracking graphs (NTGs) for component groups based on size, overlap, persistence, and level thresholds, while also compositing component images from the database into 3D renderings of the simulation (Sec. 5.3). As demonstrated in three case studies (Sec. 5.4), the generated databases grow only proportional to the parameter sampling independent of the actual simulation size, and the efficient graph operation-based NTG algorithm and image compositing procedure enable the interactive *post hoc* exploration of large-scale simulations (Sec. 5.5).

5.1 MOTIVATION

The previous chapters described a robust topology-based methodology to characterize and track features based on level, sublevel, and superlevel set components. However, applying the proposed methodology in the context of large-scale simulations containing numerous features poses additional challenges. Specifically, it is often infeasible to store every simulation state due to bandwidth and disk space constrains, and thus it is no longer possible to explicitly compute, filter, track, and render features *post hoc* for new parameters; such as different level, overlap, or persistence thresholds. This limitation necessitates *in situ* algorithms that store, at simulation runtime, the least amount of information needed to still support flexible *post hoc* analysis. Moreover, visual analytic frameworks for massive amounts of features require level-of-detail approaches that partition features into a manageable number of groups.

This chapter describes an approach that addresses these issues by combining and extending Cinema databases (Ch. 4) and nested tracking graphs (Ch. 3) for *in situ* database generation and *post hoc* database exploration, respectively (Fig. 5.1). Recall, a NTG consists of layers of common tracking graphs, where each layer visualizes the evolution of superlevel set components for a fixed level, and edges of different layers are drawn inside each other based on the nesting hierarchy of the components (Fig. 3.1). The approach presented in this chapter is based on the fact that superlevel set components merge while decreasing the level, i.e., lower layers of the NTG automatically bundle higher-level components, effectively summarizing their evolution.* Thus, instead of visualizing a cluttered view consisting of thousands of lines that represent individual features, NTGs can be used to display a limited number of lines that represent meaningful component groups. This hierarchical decomposition can also be used during the *in situ* database generation to store images of component groups instead of the individual features. As a consequence, this approach reduces the amount of stored information while still supporting common *post hoc* analysis tasks; such as toggling the visibility of groups, coloring them based on the tracking results, and linking component images and NTGs. To this end, the approach includes an efficient graph operation-based algorithm that is capable of dynamically computing NTGs *post hoc* at interactive framerates by processing split trees and other intermediate data structures that have also been stored at simulation runtime. Combining these algorithms in a feature-centric visual analytics framework enables the interactive *post hoc* analysis of large-scale simulations. The central benefit of this scalable methodology is the fact that the generated databases only grow proportional to the parameter sampling, independent of the actual geometry and number of features.

*The methodology presented in this chapter can be applied symmetrically to sublevel set components.

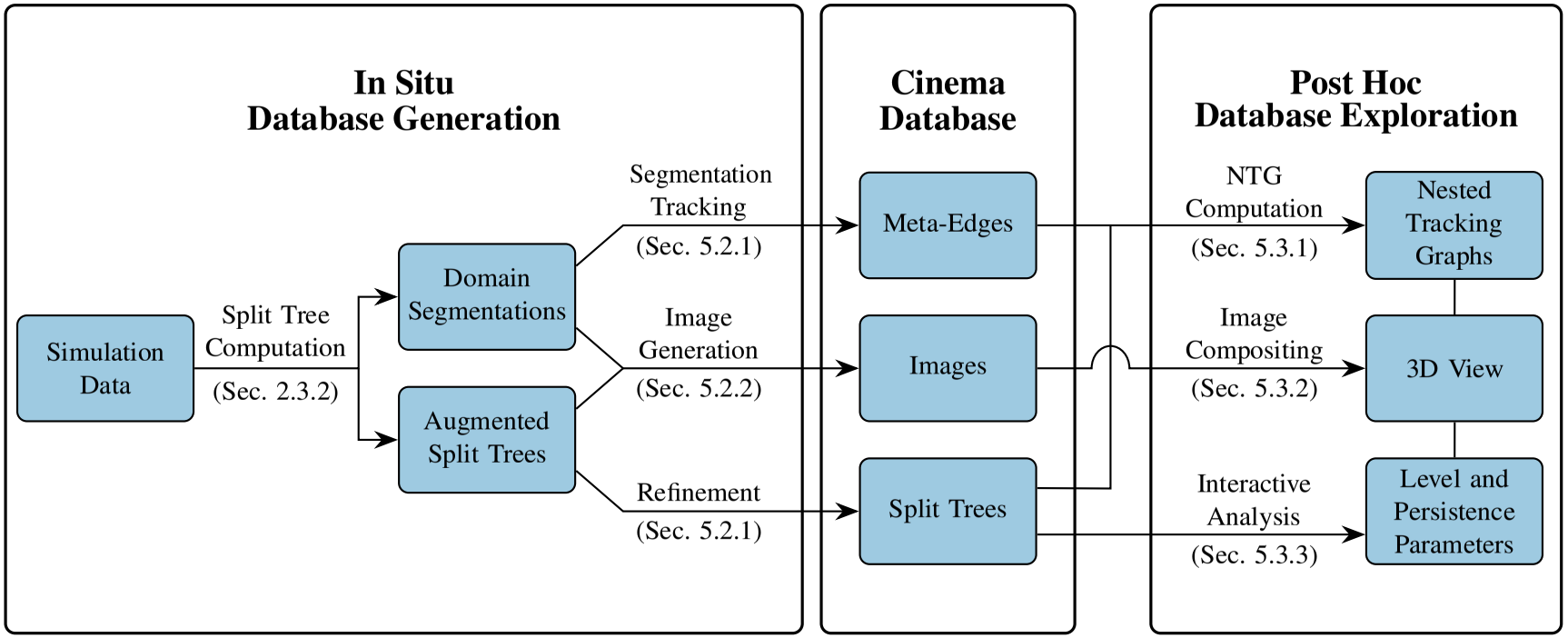


Figure 5.1: Processing pipeline of the presented approach that consists of the *in situ* database generation and the *post hoc* database exploration. During simulation runtime, the approach derives for each timestep the split tree and its associated domain segmentation to compute tracking information, images of feature groups, and refined split trees, which are stored in a Cinema database. During *post hoc* analysis, the database elements are used to dynamically compute nested tracking graphs, composite 3D views of feature groups, and to visualize the split trees and their corresponding persistence diagrams, which are all in turn integrated in a feature-centered visual analytics framework to effectively explore the underlying simulation.

5.2 IN SITU DATABASE GENERATION

During the simulation, the presented approach computes for every timestep t the complete split tree segmentation $\tilde{\mathcal{S}}_t = (\mathcal{C}_t^+, \phi_t, \psi_t)$ that consists of the split tree $\mathcal{C}_t^+$, the domain segmentation ϕ_t, and the tree scalar field ψ_t (Def. 44). Optionally, each timestep can be first simplified by persistence to remove noise (Sec. 2.3.6). These procedures are implemented in the Topology ToolKit [36, 104, 106]. The unaugmented split tree $\mathcal{C}_t^+$ and the tree scalar field ψ_t are immediately stored in a Cinema database (Sec. 2.5), whereas the augmented split tree is used to compute intermediate data structures that enable *post hoc* component tracking (Sec. 5.2.1), and to derive a reduced set of images of component groups that can later be composed again into 3D scenes (Sec. 5.2.2).

5.2.1 Merge Tree Segmentation-Based Tracking

To compute tracking graphs, it is necessary to determine the relationship between superlevel set components at adjacent timesteps. During *post hoc* analysis, it is no longer possible to explicitly compute their overlap as the volumetric data is no longer available. This triggered a line of research that aims to pre-compute tracking information to efficiently derive tracking graphs without reprocessing the original data [12, 79, 114, 119]. A prime example of such an approach is the so-called meta-graph [114] that records the overlap of component groups for discrete level intervals.

Alg. 8 is an adaption of this approach that derives edges of the meta-graph by processing two split tree segmentations $(\mathcal{C}_0^+, \phi_0, \psi_0)$ and $(\mathcal{C}_1^+, \phi_1, \psi_1)$ of consecutive PL Morse scalar fields that are defined on the same PL manifold $\mathcal{M}$.[†] Recall, a domain segmentation function ϕ_i maps any point of the domain $\mathcal{M}$ to a vertex or edge of the split tree $\mathcal{C}_i^+$, and the tree scalar field ψ_i assigns to each point on $\mathcal{C}_i^+$ the corresponding level value. Thus, these segmentations partition the domain into connected regions, called segments, that correspond to individual split tree edges (colored regions and edges in Fig. 5.2). Introducing regular vertices along tree edges further subdivides segments and will increase the tracking accuracy. Note, each split tree edge $\langle u, v \rangle \in \mathcal{C}_i^+$ with $\psi_i(u) < \psi_i(v)$ can uniquely be identified by v, and therefore v is called the edge/segment representative. This segmentation-based tracking algorithm is based on the fact that the border of a superlevel set component (dashed lines in Fig. 2.6d, left) is completely contained in the domain segment of its corresponding split tree edge (colored region and edges in Fig. 2.6c). As a consequence, if two segments overlap at a vertex v, then at least the superlevel sets for the smallest level among both corresponding intervals intersect at v. The smallest level of both intervals is therefore referred to as the base level b. Obviously, the accuracy of this

[†]This algorithm can symmetrically be formalized for join tree segmentations.

approach depends on the interval ranges, as the superlevel set at the base level represents all components for the interval of its corresponding edge. The algorithm then records the amount of spatial overlap between segments by so-called meta-edges $\mathcal{E}_M$ that connect the representatives of the corresponding split tree edges (red arrows in Fig. 5.2).

Algorithm 8: ComputeMetaEdges(PLM $\mathcal{M}$, MTS $(\mathcal{C}_0^+, \phi_0, \psi_0)$, MTS $(\mathcal{C}_1^+, \phi_1, \psi_1)$)

```
1  E_M ← ∅  // Set of Meta-Graph Edges
2  foreach vertex v ∈ M do
3      // Get edges that correspond to segments
4      (e_0, e_1) ← GetSegmentEdges( v, C_0^+, φ_0, C_1^+, φ_1 )
5      // Get edges that include base level
6      b          ← min( min ψ_0(e_0), min ψ_1(e_1) )
7      (ê_0, ê_1) ← GetBaseEdges( b, C_0^+, ψ_0, e_0, C_1^+, ψ_1, e_1 )
8      // Connect all representatives towards the root
9      AddMetaGraphEdges( E_M, ê_0, ê_1, C_0^+, ψ_0, C_1^+, ψ_1 )
10 return E_M
```

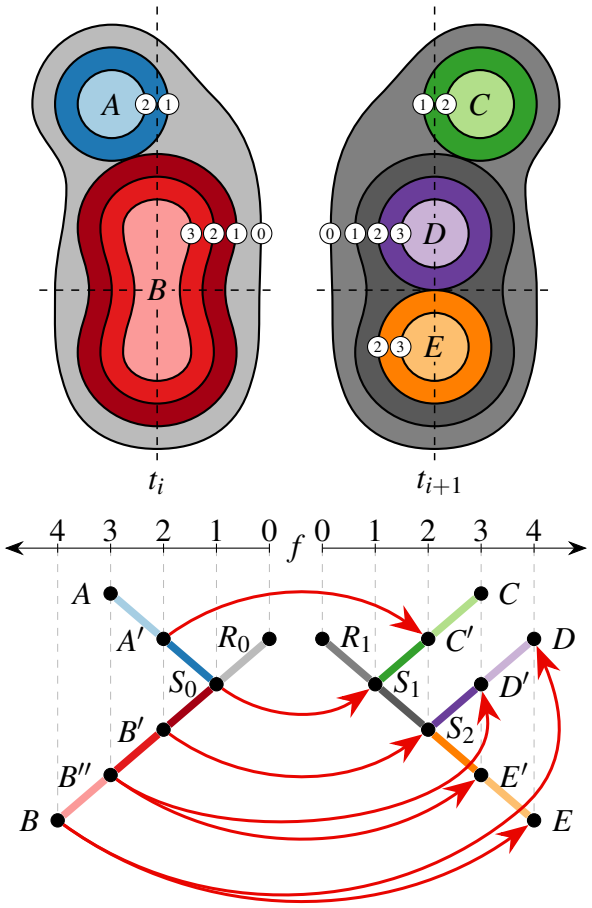


Figure 5.2: Illustration of the merge tree segmentation-based tracking approach that processes two split trees (bottom) and their respective domain segmentations (top) of two adjacent timesteps (left and right). Form time step t_i to t_{i+1}, the maximum B splits into the two maxima D and E, and the maximum A moves from the left to the right side of the domain. The overlap of segments are recorded by so-called meta-graph edges between their corresponding representatives (red arrows). For example, the dark blue and dark green segments overlap, which justifies the meta-graph edge $\langle A', C' \rangle$. The light blue and the light green segments, however, do not overlap, i.e., there exists no meta-graph edge between A and C. Note, the meta-graph edges correctly record the evolution of the overlapping segments across all intervals. Yet, the accuracy of the matching depends on the resolution of the intervals.

Specifically, Alg. 8 initializes the meta-edges $\mathcal{E}_M$ as an empty set, and then processes each vertex $v \in \mathcal{M}$ in three steps. First, line 4 retrieves for a vertex v the corresponding edges $e_0 \in \mathcal{C}_0^+$ and $e_1 \in \mathcal{C}_1^+$ (thick edges of Fig. 5.3b) with the domain segmentation functions ϕ_0 and ϕ_1. Note, these edges do not have to correspond to the same level interval. As explained earlier, it can only be guaranteed that the respective superlevel sets for the base level intersect at v, where the base level b is the minimum of both level intervals associated with e_0 and e_1. Next, it is necessary to find the edges $\hat{e}_0 \in \mathcal{C}_0^+$ and $\hat{e}_1 \in \mathcal{C}_1^+$ whose respective intervals include the base level. This is done with the procedure *GetBaseEdges* that traverses each split tree $\mathcal{C}_i^+$ starting at the edge e_i towards the root until it finds and returns the first edge whose interval includes the base level b (thick edges of Fig. 5.3c). With the same argument as before, it is guaranteed that components of the base edges overlap for the base level, and therefore the procedure *AddMetaGraphEdges* adds a meta-edge between their representatives (red arrow in Fig. 5.3c). Furthermore, if features of these edges overlap, then also do the features of the edges towards the root. Therefore, the procedure *AddMetaGraphEdges* synchronously traverses both trees towards the root and adds meta-graph edges (red arrows in Fig. 5.3d) between the representatives of the visited edges (thick edges in Fig. 5.3d). This procedure can also additionally record the amount of spatial overlap between segments, e.g., by counting how often a meta-graph edge would have been added during all iterations. Finally, the algorithm returns the set of meta-graph edges between the two segmentations.

This procedure can be executed iteratively for each adjacent pair of a PL Morse scalar field sequence to derive the complete set of meta-graph edges, and in an *in situ* environment it is only necessary to keep the segmentations of the previous and current timestep in memory. The meta-graph then corresponds to the union of all split trees and meta-graph edges. The described procedure is implemented in the *TrackingFromMergeTreeSegmentations* module [61] of the *Topology ToolKit* [104], which was used for all experiments described in Sec. 5.4.

Meta-graphs enable the efficient *post hoc* computation of tracking graphs for any level l, solely based on their structure and the tree scalar fields. Specifically, a trivial algorithm first retrieves the set of split tree edges of the meta-graph whose intervals include l, and creates for each such edge a vertex in the resulting tracking graph. These vertices are subsequently connected based on the corresponding meta-graph edges that belong to the representatives of the split tree edges. Sec. 5.3.1 describes a graph operation-based algorithm that derives nested tracking graphs.

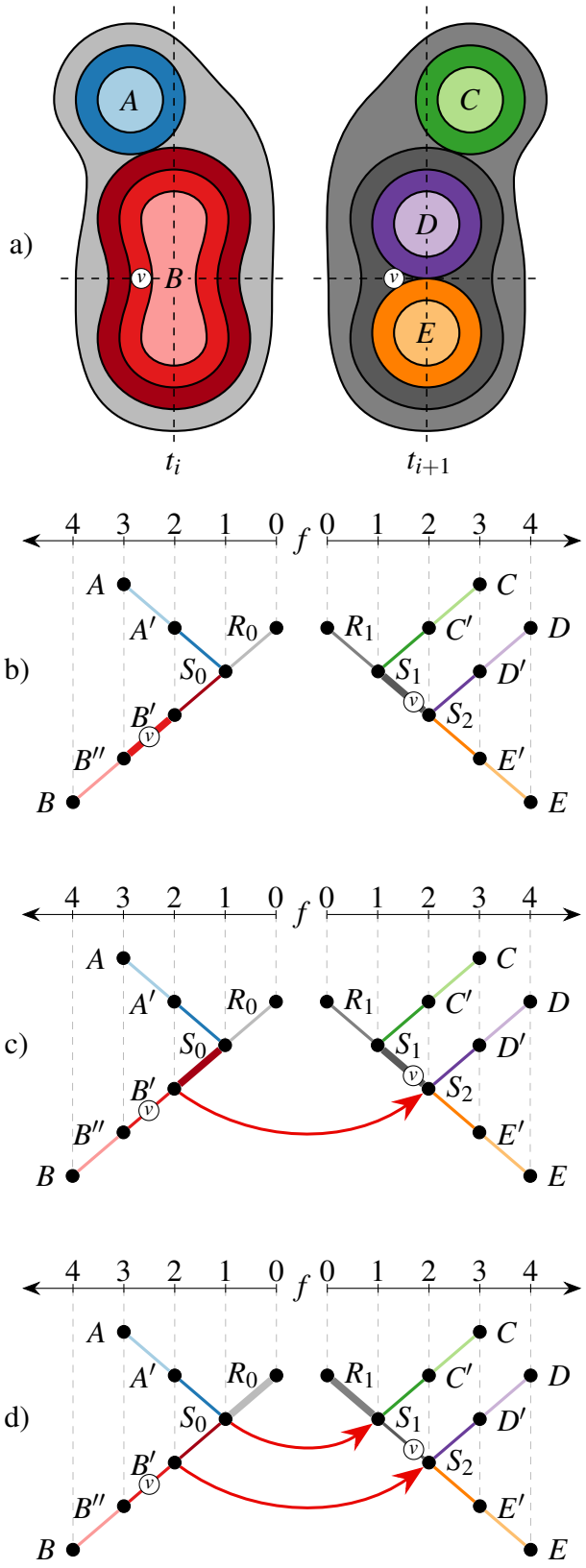


Figure 5.3: Illustration of one inner iteration of Alg. 8 that processes a vertex v of the example scalar field introduced in Fig. 5.2. First, the algorithm retrieves for vertex v the corresponding edges of the split trees (thick edges in Figure b). Next, it traverses both trees towards the root until it finds the base edges (thick edges in Figure c), i.e., edges towards the root whose corresponding intervals include the base level (in this example 1). In step three, the algorithm adds a meta-graph edge between the representatives of the base edges (red arrow in Figure c), and then continuous traversing the trees towards the root while also connecting the representatives of the visited edges (Figure d). Note, this correctly records the possible overlaps of superlevel sets that include vertex v. Specifically, superlevel sets for level 1 overlap at v (as indicated by the meta graph edge $\langle B', S_2 \rangle$), but not for level 2 (as indicated by the absence of a meta-graph edge from B'' to D' or E'). Iterations processing the remaining vertices add additional meta-graph edges, or increase the incidences of existing ones. For instance, a vertex located in the dark purple segment introduces the meta-graph edge $\langle B'', D' \rangle$, and increases the overlap of meta-graph edges $\langle B', S_2 \rangle$ and $\langle S_0, S_1 \rangle$. This again shows that the accuracy of the segmentation matching depends on the resolution of the level intervals.

5.2.2 Image Generation

To provide an interactive 3D rendered view of the simulation *post hoc*, the proposed approach also stores, at simulation runtime, images of superlevel set component groups that can later be composed again into 3D scenes. The following algorithm is built on top of the original Cinema approach [2] that generates images for a Cartesian product of the parameter space (Sec. 2.5). Database viewers then enable users to browse the structured image stores by selecting interesting parameter combinations from parallel coordinate plots [115], by performing queries [3, 104], or by snapping to the closest available camera locations while navigating an emulated 3D view [82]. The approach described in Ch. 4 can even compute a reduced set of images that enable free camera movement by approximating the depicted surfaces. However, a limitation of Cinema databases is that the flexibility of the *post hoc* analysis is limited by the generated images. Thus, if the database does not contain individual feature images, it is not trivially possible to toggle their visibility. Storing an image of each feature is also problematic as this drastically increases the amount of database elements. Therefore, it is necessary to depict feature in groups with common *post hoc* analysis tasks in mind.

In the context of tracking superlevel set components in large-scale simulations, analysts should be able to toggle the visibility of components that are locally clustered together, and further filter components based on persistence. To this end, the proposed approach partitions components into a predefined number of groups based on a branch decomposition $\mathcal{B}$ of the current split tree $\mathcal{C}_t^+$, and a list of persistence intervals P. The algorithm then generates images for each resulting component group. Specifically, the inputs of Alg. 9 are PL manifold $\mathcal{M}$, its PL scalar function f, a split tree segmentation $(\mathcal{C}_t^+,\ \phi_t,\ \psi_t)$, a set of camera specifications C, a set of levels L, a sorted list of persistence thresholds P, and the maximum number of component groups n; i.e., each timestep yields at maximum $|C| \cdot |L| \cdot |P| \cdot n$ images. First, the algorithm sorts all branches by persistence in descending order, and then inserts the n most persistent branches into their own new group (line 1-6). Each remaining branch is then inserted into the group that contains the most persistent branch it is attached to (lines 7-11). Note, such a branch and the corresponding group must exist as the branches are processed in sorted order.

Next, the algorithm iterates over the groups $G \in \mathcal{G}$, and the persistence intervals defined by P, to determine in each iteration the branches $\hat{\mathcal{B}} \subseteq G \in \mathcal{G}$ inside the current persistence interval $(P_i, P_{i+1}]$. Then, the algorithm derives for each level $l \in L$ the set of individual contours X of the current group, i.e., the borders of the superlevel set components. This is done by first determining the branches that include the current level, where each such branch B indicates the existence of an individual superlevel set

component. To derive the set of simplices $\hat{\mathcal{M}} \subseteq \mathcal{M}$ that together completely contain the component of B, the algorithm first collects the set of edges $\hat{\mathcal{C}}^+$ that are connected to B above the current level (the upper subtree of B that exceeds the level), and then retrieves all simplices of $\mathcal{M}$ that share at least one vertex with the subtree domain $\phi^{-1}(\hat{\mathcal{C}}^+)$. It is necessary to include the tetrahedra adjacent to the subtree domain as they might contain parts of the linearly-interpolated contours.

Algorithm 9: GenerateImages($\mathcal{M}$, f, $\mathcal{C}_t^+$, ϕ_t, ψ_t, C, L, P, n)

```
// Get branches sorted by persistence in descending order
B ← ComputeBranchDecomposition( C+, ψ_t )
// Create groups for the first n most persistent branches
G ← NewUnionFind()
for i ← 0 to n − 1 do
    NewGroup( G, B_i )
// Add remaining branches to closest group
for i ← n to |B| do
    B ← GetMostPersistentAttachedBranch( B, B_i, ψ_t )
    G ← FindGroup( G, B )
    AddToGroup( G, B_i )
// Generate group images for all persistence intervals and levels
foreach group G ∈ G do
    foreach threshold p_i ∈ P where p_i ≠ max(P) do
        // Filter grouped branches by persistence
        B̂ ← { B ∈ G | p_i < (max ψ_t(B) − min ψ_t(B)) ≤ p_{i+1} }
        foreach level l ∈ L do
            // Add contour for each filtered branch that includes level
            X ← ∅  // Set of contours
            foreach B ∈ B̂ where min ψ_t(B) < l ≤ max ψ_t(B) do
                Ĉ+ ← GetUpperTreeOfBranch( B, C_t^+, ψ_t, l )
                M̂ ← { σ ∈ M | σ ∩ φ_t^{-1}(Ĉ+) ≠ ∅ }
                AddContour( X, M̂, f, l )
            // Render depth and ID image of contours for each camera
            foreach camera c ∈ C do
                I ← RenderContours( X, c )
                StoreInCinemaDB( I, G, p_i, p_{i+1}, l, c )
StoreInCinemaDB( G, t )
```

Finally, the algorithm renders for all camera angles C a depth image and an ID mask of all contours (Fig. 5.4), where the depth images are used during *post hoc* analysis to compose 3D views (Fig. 5.5), and a pixel of the ID mask stores the representative of the split tree edge that corresponds to the depicted contour. The images are then stored in the Cinema database, where they are also associated to the parameters that uniquely identify the images: their group ID, persistence interval, level, and camera angle. To efficiently retrieve during *post hoc* analysis an image that depicts a specific contour, the algorithm also stores, in line 28, the branch groups $\mathcal{G}$ of the current timestep in the Cinema database.

Note, the image generation is embarrassingly parallel as images for component groups and camera angles can be rendered independently. A limitation of this approach is that the sampling resolution of the parameter space is directly proportional to the resulting image database size. Moreover, the parameter sampling has to be determined beforehand, in which case adequate parameters might be unknown.

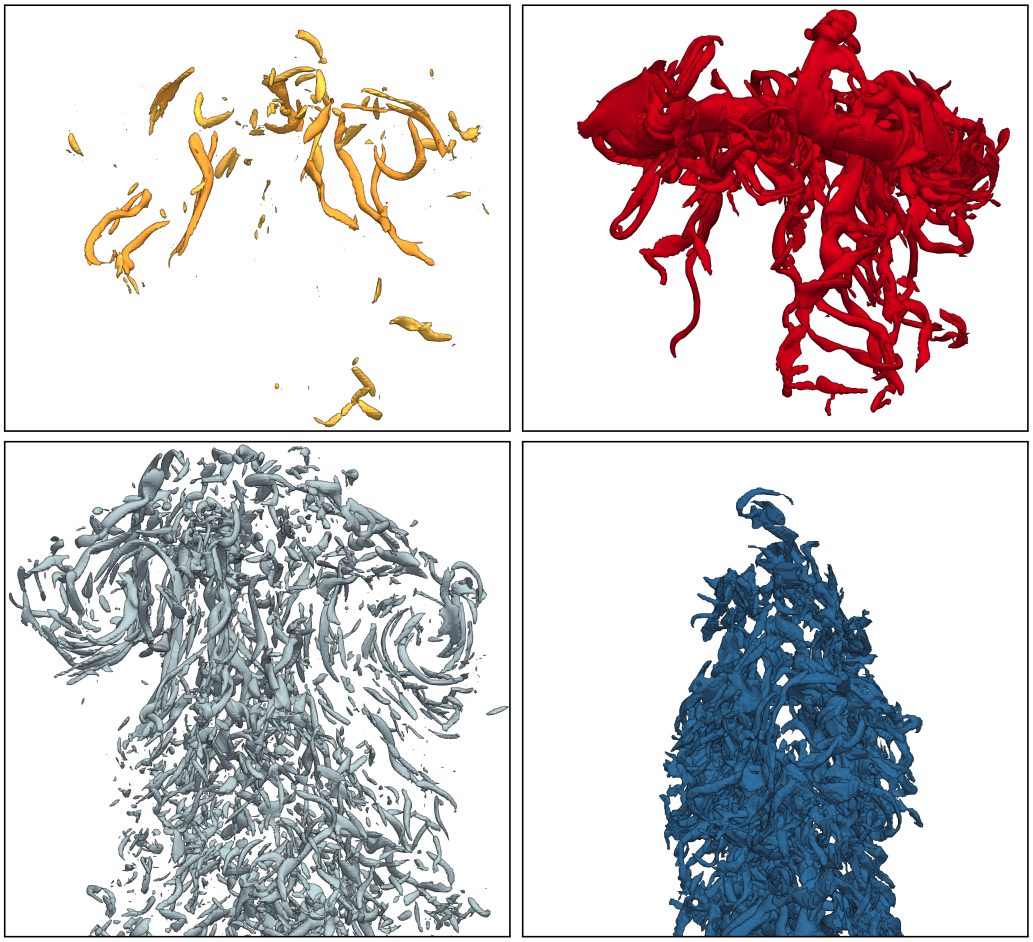


Figure 5.4: Four generated images of $\sim 5k$ vortices from the jet dataset at timestep 2000 based on two groups (top and bottom) and two persistence intervals (left and right).

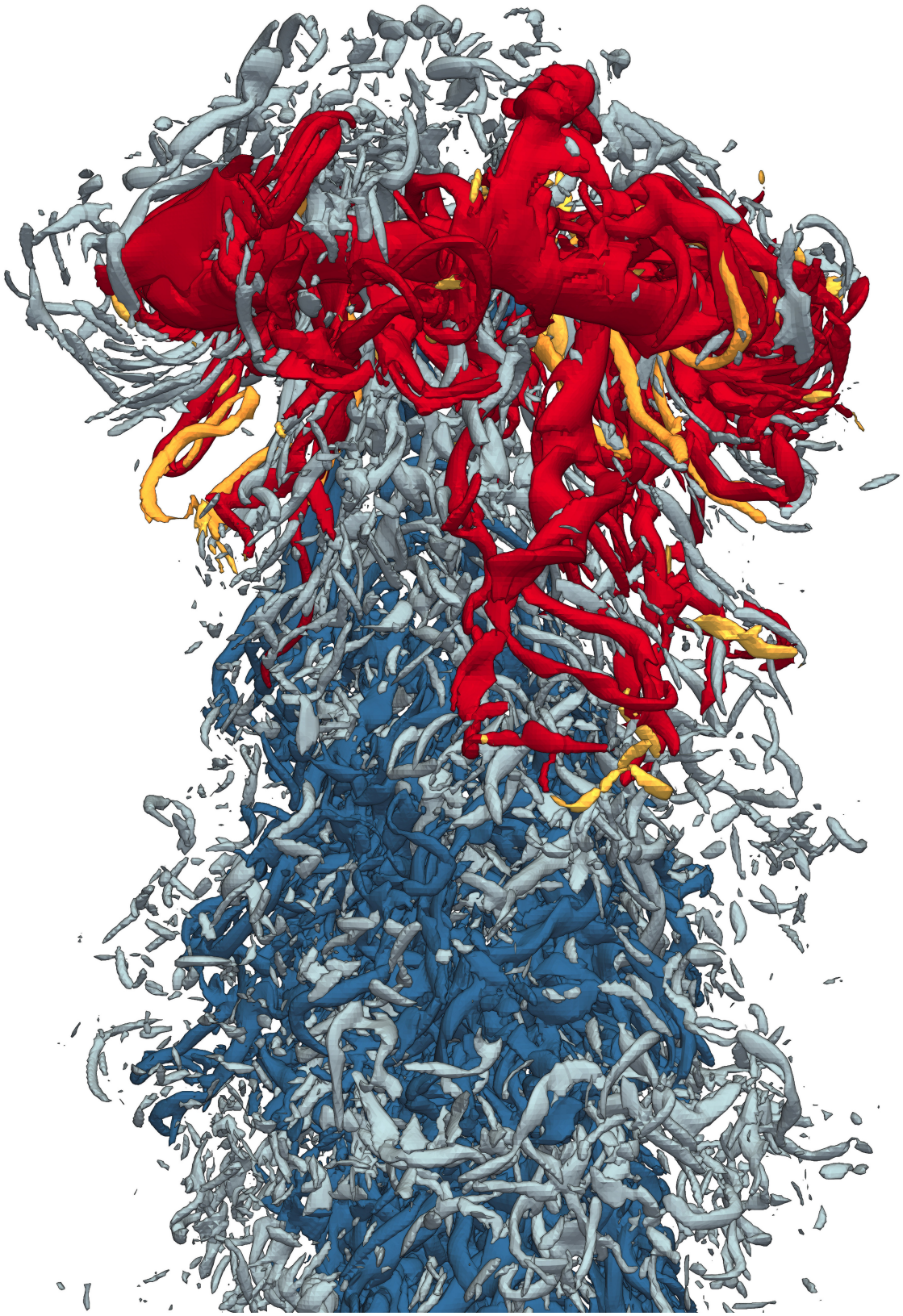


Figure 5.5: 3D view of the jet dataset composited with the four images of Fig. 5.4.

5.3 POST HOC DATABASE EXPLORATION

This section describes the novel graph operation-based NTG algorithm (Sec. 5.3.1), the image compositing pipeline (Sec. 5.3.2), and the visual analytics framework (Sec. 5.3.3) that all use the generated Cinema database to analyze the underlying simulation *post hoc*.

5.3.1 Dynamic Nested Tracking Graphs

The core element of the *post hoc* analysis interface is a NTG that enables users to browse the simulation data across time and levels. Computing the NTG with the original procedure described in Alg. 5 would make it necessary to predefine a set of levels, explicitly compute the superlevel set components for those levels, and then test the resulting components for spatial overlaps across time (to determine their evolution) and across levels (to determine their nesting hierarchy). Thus, changing the levels requires the re-computation of all components, which is inefficient and unsuitable for large-scale simulations and *in situ* use cases.

Alg. 10 outlines a purely graph operation-based NTG algorithm that efficiently computes NTGs for a sorted list of adjacent timesteps T, a sorted list of levels L, and the graph structures that have been stored in the Cinema database at each timestep during the simulation: the split trees $\mathcal{C}^+$, their scalar functions ψ, and the meta-edges $\mathcal{E}_M$. First, the algorithm determines the superlevel set components that are present for all timesteps and levels based only on the split trees $\mathcal{C}^+$ and their corresponding scalar fields ψ. Given a timestep $t \in T$ and a level $l \in L$, the algorithm inserts a new vertex into the set $\mathcal{V}$ for each edge $\langle u, v \rangle \in \mathcal{C}_t^+$ whose corresponding level interval includes l, as each such an edge represents an individual superlevel set component (red vertices in Fig. 5.6). In the following, each vertex of $\mathcal{V}$ is denoted as v_t^l to compactly indicate its corresponding timestep t, level l, and edge representative v in the split tree $\mathcal{C}_t^+$. The nesting hierarchy $\mathcal{E}_N$ (red edges in Fig. 5.6) of the computed vertices $\mathcal{V}$ follows immediately from the structure of the split trees (black edges in Fig. 5.6). To identify the connections between vertices at level $l_i \in L$ for $i > 0$ (children) with vertices at level $l_{i-1} \in L$ (parents), the algorithm simply traverses the tree from each child towards the root until it encounters a parent and then inserts a new edge into $\mathcal{E}_N$ accordingly. Since the algorithm descends in a rooted tree, there always exists exactly one parent for each child. Finally, the algorithm needs to establish the relationships between vertices at the same level for adjacent timesteps t and $t+1$. This can be done efficiently via the meta-edges $\mathcal{E}_{M,t}$ of timestep t. Specifically, for each two vertices u_t^l and v_{t+1}^l one can determine if the segments that are represented by u and v overlap by checking if $\mathcal{E}_{M,t}$ contains the meta-edge $\langle u, v \rangle$. If it does, the algorithm adds the edge $\langle u_t^l, v_{t+1}^l \rangle$ to $\mathcal{E}_T$. It is possible to filter tracking graph edges via an overlap

threshold, or relax the tracking accuracy by adding edges if there exists an meta-edge for a vertex pair further down in the split tree. Such a relaxation enables the tracking of fast moving components whose corresponding segments only overlap for lower levels. The advantage of the proposed algorithm is that such criteria can be interactively chosen *post hoc* without access to the original simulation data. That the computed graph is indeed a NTG according to Def. 50 follows directly from Alg. 10, and the fact that the subprocedure *AddMetaGraphEdges* of the segmentation-based tracking algorithm (Alg. 8) recursively adds meta-edges between overlapping segments (Fig. 5.3d). The resulting NTG can be rendered with Alg. 6. The graph operation-based NTG algorithm is implemented in the *NestedTrackingGraph* module [61] of the *Topology ToolKit* [104].

Algorithm 10: ComputeNTG(Times T, Levels L, Split Trees $(\mathcal{C}^+, \psi)$, Meta-Edges $\mathcal{E}_M$)

```
1  V, E_N, E_T ← ∅   // Vertices, Nesting Trees, Tracking Graphs
2  // Compute Vertices
3  foreach timestep t ∈ T do
4      foreach level l ∈ L do
5          foreach edge ⟨u,v⟩ ∈ C_t^+ where ψ_t(u) < l ≤ ψ_t(v) do
6              AddVertex( V, v, l, t )
7  // Compute Nesting Trees
8  foreach vertex v_t^l ∈ V where l ≠ min(L) do
9      AddEdge( E_N, v_t^l, GetParent(v_t^l, V, C_t^+, ψ_t) )
10 // Compute Tracking Graphs
11 foreach vertex u_t^l ∈ V where t ≠ max(T) do
12     foreach vertex v_{t+1}^l ∈ V do
13         if ⟨u,v⟩ ∈ E_{M,t} then
14             AddEdge( E_T, u_t^l, v_{t+1}^l )
15 return V ∪ E_N ∪ E_T
```

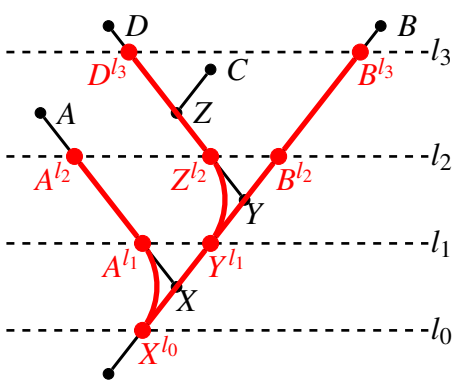


Figure 5.6: Vertex and nesting tree computation based on split trees. Vertices correspond to edge-cuts (red nodes) for a set of levels (dashed lines), where each vertex represents a single superlevel set component, and is labeled by its corresponding edge representative, level, and timestep (here omitted). To determine their nesting hierarchy (red edges), the algorithm traverses the split tree from each vertex at level l_i with $i > 0$ towards the root, until the algorithm reaches its parent at level l_{i-1}.

5.3.2 Image Retrieval and Compositing

To retrieve the image of a component corresponding to a vertex $v_t^l \in \mathcal{V}$ of the NTG for a specific camera angle, one first determines its branch group $G \in \mathcal{G}_t$, and then computes the persistence interval of the branch containing the edge represented by the vertex $v \in \mathcal{C}_t^+$ (Sec. 5.2.2). All parameters are then used to retrieve the closest available image in the database. Fig. 5.7 illustrates the Depth Image Based Rendering (DIBR) pipeline that composes multiple depth images and ID masks into a single image. To improve spatial perception, the images are shaded based on approximated surface normals and screen space ambient occlusion, where components are colored based on the ID masks. Alternatively, it is also possible to use the geometry approximation algorithms described in Sec. 2.6 and Ch. 4 to enable free camera movement.

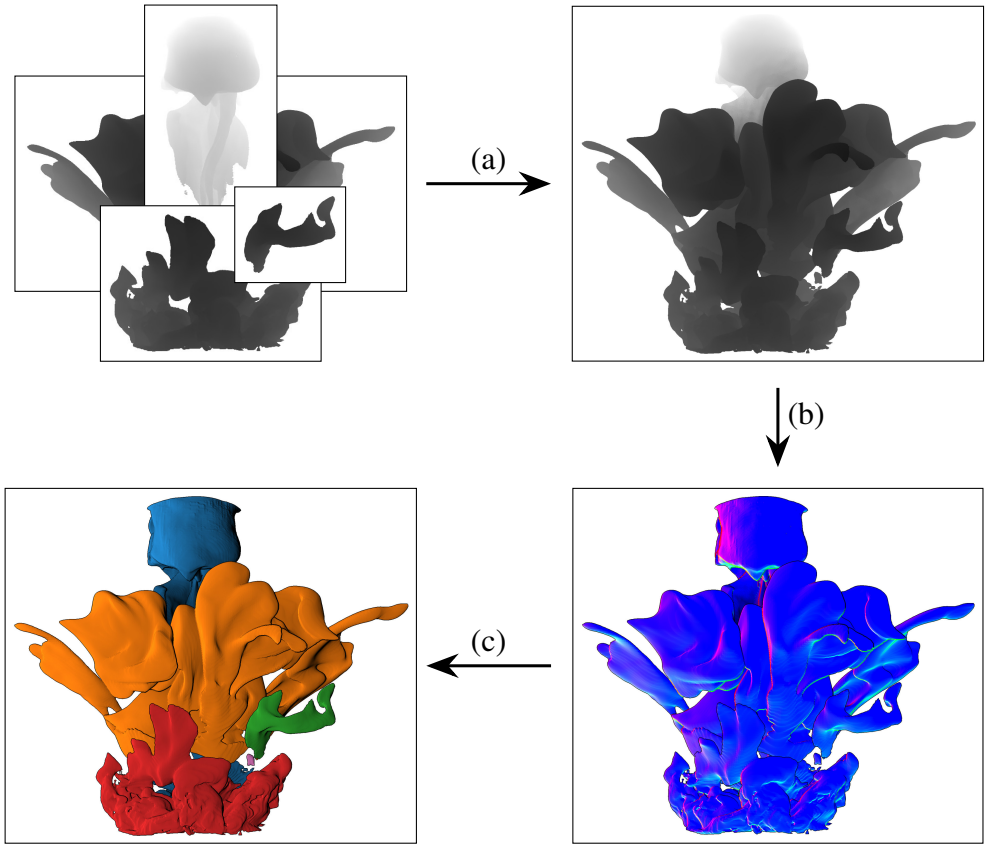


Figure 5.7: Depth image-based rendering pipeline: multiple depth images and ID masks are respectively composed into a single image (a), which are shaded based on approximated surface normals (b) and screen space ambient occlusion (c) in order to improve spatial perception.

5.3.3 Visual Analytics Framework

Fig. 5.8 shows all linked components of the *post hoc* visual analytics framework that enables users to effectively explore the generated Cinema database: a composed 3D scene (top left), a split tree (top center), a persistence diagram (top right), and a nested tracking graph (bottom). User interface (UI) elements that correspond to an individual superlevel set component are consistently colored across all views, i.e., edges of the NTG, images of the components, branches of the split tree, and critical-point pairs of the persistence diagram. The core element of the interface is the NTG that illustrates the evolution of components for multiple levels, whereas the split tree shows their nesting hierarchy for the current timestep, and the persistence diagram shows their significance. The NTG is used to select time intervals, individual timesteps, and specific components, and the split tree and persistence diagram support analysts in choosing appropriate levels and persistence thresholds. The current persistence threshold is drawn as a diagonal red line in the persistence diagram, and levels of the NTG are drawn as horizontal lines in the split tree and persistence diagram, where the line of the currently selected level is also colored red. The 3D view is composed of images that are closest to the current parameter settings, i.e., the closest available database elements for a requested view angle, persistence interval, and selected level. Components that do not exists for a selected level or that do not exceed a persistence threshold are grayed out in all views.

The interface provides three key mechanisms to handle numerous components: 1) before parameter updates the interface indicates the resulting numbers of components, split tree branches, and NTG edges; 2) components can be filtered based on size, persistence, and overlap thresholds; and 3) if numerous components have been chosen for visualization, the interface initially groups them together based on the nesting hierarchies and persistence values to generate a manageable amount of UI elements. Specifically, instead of rendering the entire split tree at once, the interface initially draws only a user-controlled number of the most persistent branches. Analysts then have the option to further expand individual branches, where the number of children is encoded by the width of the parent branch. Similarly, instead of rendering numerous tracks of the NTG for a certain level, these tracks are initially represented by their parent edges at the lower layers, and analysts can interactively toggle their visibility.

Layout updates of the graphs are only performed when necessary, or on request. For example, tightening the thresholds filters more components, which results in less NTG edges, split tree branches, and critical point pairs. Instead of updating the graph layouts immediately, the corresponding UI elements are simply removed, so that analysts can easily comprehend the updates without reorienting themselves within a new layout

(Fig. 5.9 middle and bottom). However, analysts always have the option to recompute the layouts while ignoring the filtered components to generate smoother graphs. The interface also provides visual consistency when a new level is added to the NTG. Specifically, the layout algorithm described by Lukasczyk et al. [66] processes the layers of the NTG individually, and then stacks them in a bottom-up approach. Hence, inserting a level does not effect the layers of levels smaller than the new level, i.e., adding a level that is larger than all current levels results in a new layer that is completely embedded in the previous top layer. Overall, the interface enables analysts to follow the history of individual components and groups, filter them based on various metrics, and explore the simulation in a focus+context approach.

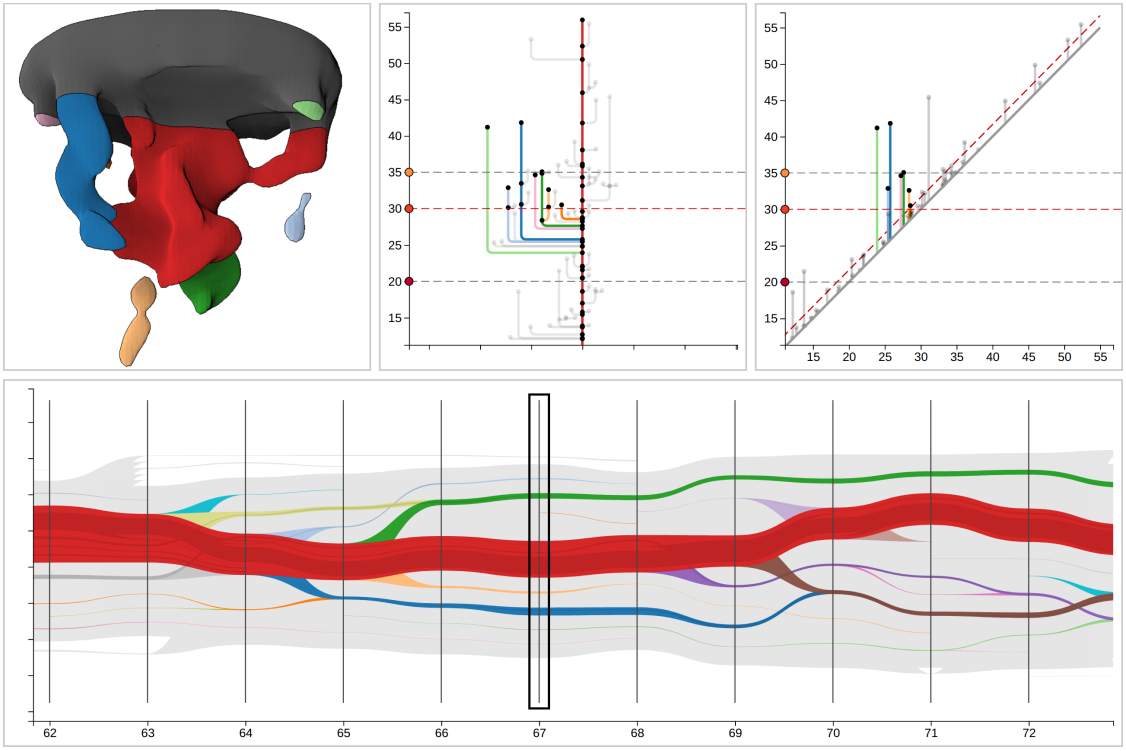


Figure 5.8: The presented topology-based visual analytics framework supports feature-centered navigation of Cinema databases consisting of image and analysis products generated during large-scale simulation runs. Here, the interface shows an ensemble member of the viscous finger dataset for a salt concentration level of 30, where colors correspond to individual fingers. The prime interaction device of this interface is a nested tracking graph (NTG) that displays the temporal evolution of superlevel set components and their properties for multiple levels simultaneously (bottom). The NTG is used to retrieve component images that are stored in a Cinema database (top left), whereas the split tree (top center) and persistence diagram (top right) support users in selecting important levels and filter criteria, which are in turn used to update the NTG in real-time.

5.4 RESULTS

This section evaluates the proposed methodology based on three real-world examples. The first case study compares the *post hoc* tracking algorithm to the explicit approach described in Sec. 3.2.2 by contrasting the resulting graphs for the 2016 scientific visualization contest dataset [42] (Sec. 5.4.1). To demonstrate that the proposed approach can be used to effectively explore large-scale simulations with numerous components, the other two case studies deal with much larger and more complex datasets—i.e., the simulation ensemble of the 2018 scientific visualization contest [43] (Sec. 5.4.2), and a computational fluid dynamics simulation with thousands of vortex features (Sec. 5.4.3).

5.4.1 Viscous Fingering

This case study compares the results of the graph operation-based tracking approach and the original overlap-based algorithm (Sec. 3.2.2) for the viscous fingering simulation ensemble that was already introduced in Sec. 2.4.2. In a nutshell, the simulations model the process of viscous fingering inside a water filled cylinder with an infinite salt supply at its top (gray surface in Fig. 5.8). As the salt mixes with the water, the solutions form characteristic structures with increased salt concentration values, called viscous fingers (colored components in Fig. 5.8), which can be identified algorithmically by first sampling the salt concentration density of the pointsets on a regular grid and then deriving superlevel set components below the salt supply (Sec. 2.4.2).

The top and middle row of Fig. 5.9 show two NTGs for the same simulation run, where the first graph is derived with the original approach that explicitly computes the overlap of superlevel set components (Alg. 5), and the second graph is derived with the graph operation-based algorithm that processes meta-edges and split trees (Alg. 10). The graphs mostly match, except that the new algorithm adds more edges than the original approach. This is due to the segmentation-based tracking approach, as components inside a segment are collectively tracked based on the largest component (Sec. 5.2.1). Thus, the new algorithm detects at least the same amount of overlaps as the old approach, but also matches components whose corresponding segments overlap. For instance, the volumes of fast moving components might not overlap in time, and therefore the original algorithm identifies the components in each timestep as new emerging features, which is semantically incorrect. However, the corresponding domain segments are likely to overlap since they correspond to the same moving maximum. As a consequence, the segmentation-based algorithm identifies the components as a single moving feature (thin lines of Fig. 5.9). Choosing an appropriate segmentation refinement level during the meta-edge generation improves the accuracy of this matching (Sec. 5.2.1). In all presented

experiments, this refinement level was set to the persistence threshold that was used to remove noise, which yielded adequate results. In fact, choosing the refinement level in this way produces the same NTG as the explicit approach for the example shown in Fig. 5.9. The segmentation-based algorithm makes it also possible to interactively restrict or relax tracking criteria by respectively requiring a minimum amount of overlap, or by additionally matching segments that are connected via meta-edges further down in trees.

The main advantage of the segmentation-based algorithm is that once the meta-edges have been computed, the NTG algorithm no longer requires access to the volumetric simulation data. Processing the meta-edges and split trees is also significantly faster than explicitly computing superlevel set components and their respective overlaps: deriving NTGs for one ensemble member for the same parameters on the same hardware takes on average ~ 6 seconds with the old approach, and ~ 0.1 seconds with the new algorithm. NTGs for the following jet and asteroid case studies can still be computed in milliseconds, whereas the explicit approach requires several minutes. This speedup enables analysts to interactively update level, persistence, overlap, and size constraints (Fig. 5.9). To summarize, the *post hoc* tracking algorithm is capable of tracking features more accurately and flexible than the explicit approach, and enables users to compute NTGs in real-time.

Obviously, an image database for such a small dataset requires far more disc space than the original data (Tab. 5.1). In fact, storing images become only beneficial for extremely large datasets, since the primary advantage of an image database is that its size grows proportional to the parameter sampling, independent of the size of the depicted simulation [2]. This can be observed in all presented experiments.

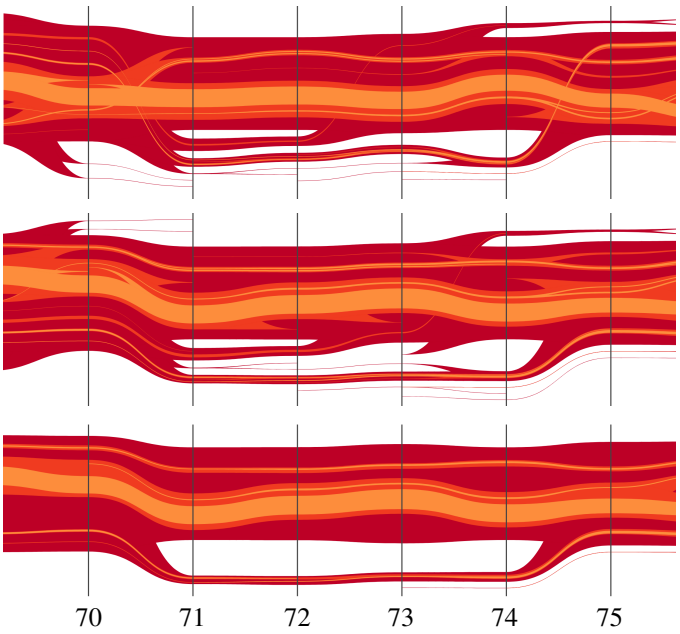


Figure 5.9: NTGs of the viscous finger dataset for salt concentration levels 25, 30, and 35 (red to yellow). (Top) NTG generated with the explicit overlap-based tracking approach outlined in Alg. 5. (Middle and Bottom) NTGs generated with the split tree-based tracking approach outlined in Alg. 10, where the bottom graph is filtered by persistence, size, and overlap thresholds.

5.4.2 Asteroid Impacts

This case study examines an ensemble of extreme-scale simulations that are part of a threat assessment study of asteroid ocean impacts [43, 85], where individual ensemble members correspond to various impact scenarios based on different asteroid sizes, impact angles, and airburst heights. Each simulation is labeled according to the following convention: the first letter is an ensemble index, the second letter corresponds to the airburst height above sea level (A: None, B: 5km, and C: 10km), the third letter represents the asteroid diameter (1: 100m, 3: 250m, and 5: 500m), and the fourth letter indicates the impact angle (0: 27.4°, 1: 45°, and 2: 60°). As oceans cover around 71% of Earth's surface, they are the most likely location of an asteroid impact. Therefore, the main objective of the threat assessment is to explore the relationship between the impact scenarios and the severeness of the tsunami they create upon impact. For instance, providing a minimum asteroid size threshold would greatly support the effort of NASA's Planetary Defense Coordination Office [76] in tracking potentially dangerous objects. The following case study will demonstrate that the proposed approach enables analysts to efficiently explore and compare these different impact scenarios.

The original simulations advance an Eulerian grid that is adaptively refined at significant areas based on the xRAGE simulation code [35]. The simulations compute, among others, a temperature field on a regular grid with either 300^3 or 500^3 vertices. To generate a Cinema database according to the proposed approach, these temperature fields are streamed into an emulated *in situ* environment that processes each timestep. Tab. 5.1 shows the total computation time and size of analysis and image products on a cluster node with an *Intel E5-2640v3* processor (16 cores) and 256GB memory. The stated time measurements include the computation of split tree segmentations [36], topological simplifications [106], and meta-edges. Note, the image generation process is embarrassingly parallel, so the actual image generation time is much lower in practice. The provided image database sizes correspond to a sampling at 24 cameras, 6 levels, 2 persistence intervals, and 4 component groups, which enables users to adequately rotate the 3D view and update parameters. Although the size of simulation *yA*31 is almost five times bigger on the 500^3 grid than on the 300^3 grid, their respective image databases are roughly the same size since components are depicted in a fixed number of groups. This demonstrates that the database size is decoupled from the size of the underlying data, but to provide more flexibility it is necessary to sample the parameter space more thoroughly. Thus, the proposed approach can scale to very large data sizes with an acceptable flexibility trade-off during *post hoc* analysis.

Fig. 5.10 shows for timestep 108 of simulation run *yA*31 (500^3) the split tree (third row), a composited 3D view (fourth row), and an NTG that is once colored by layer (first row), and once colored by individual components for level $0.2eV$ (second row). Here, the NTGs clearly illustrates that at the time of impact the entire region around the impact site is a single burning volume that disperses over time into four sub volumes (blue, red, orange, and green UI elements). Since NTGs can be updated coherently in real-time and since their layers partition components into groups, analysts can interactively explore different levels and the corresponding components by expending edges of the split trees and NTGs. With this focus+context approach, analysts can select individual components and their respective tracks for detailed examination, and it is possible to further filter the graph based on size, overlap, and persistence thresholds. Additionally, the split tree and persistence diagram support the user in selecting important parameters. Based on a component selection, the interface then queries and composes images form the Cinema database into a 3D rendering of the simulation. Note, although the database contains only two persistence intervals, filtered components can at least be colored gray in the composited view. Thus, if there exists a small number of low persistent components—i.e., they do not clutter the 3D view—then a small number of persistence intervals is sufficient. All interface elements together then guide the user while examining specific components or component groups; e.g., the split tree indicates that the green component contains the global maximum, whereas the NTG and the 3D view show that the volume of the green component is relatively small. The low overhead of the graph operation-based NTG algorithm and the image compositing enables analysts to quickly update parameters and cycle through different ensemble members at interactive framerates. This is the prime advantage over previous approaches.

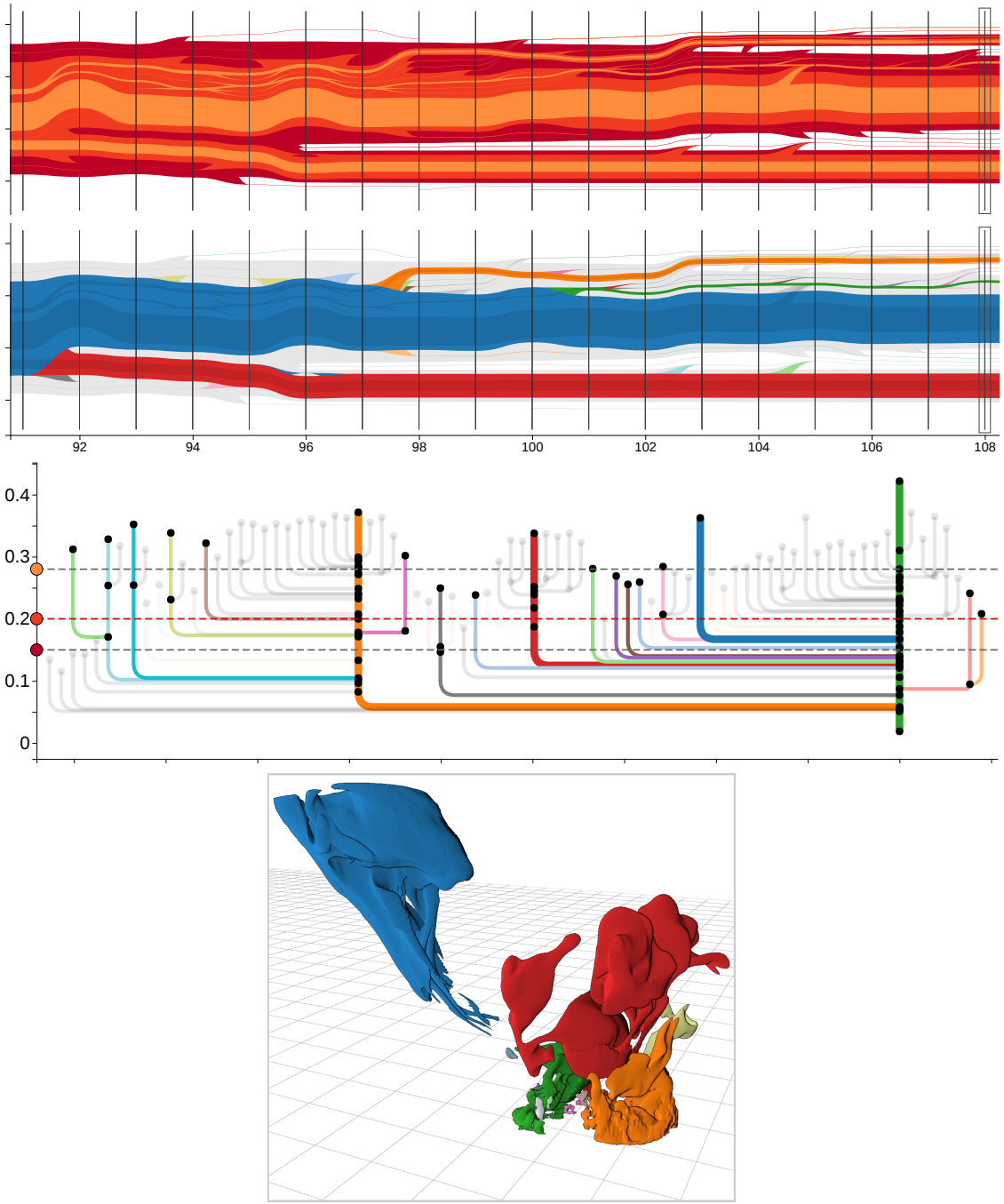


Figure 5.10: Analysis of the asteroid impact yA31 temperature field. First and second row: NTGs visualizing the evolution of the temperature field for the levels $0.15eV$, $0.2eV$, and $0.28eV$, where the second graph highlights level $0.2eV$. Third and fourth row: Split tree and 3D view of timestep 108, where the tree indicates the nesting hierarchy of features, and 3D view composes images for level $0.2eV$. The proposed approach partitions the temperature volumes—and their contained subfeatures—into groups: the asteroid trail (dark blue), the cloud that raises to the stratosphere (dark red), the wave that thrusts forward over the ocean (dark orange), and the volume containing the global maximum at the impact site (dark green).

5.4.3 Jet Simulation

This case study focuses on a jet simulation and was chosen to illustrate the utility of the proposed approach for feature-rich datasets. The simulation describes a high-velocity fluid jet entering a medium at rest. Due to viscous effects, a large vortex ring is generated at the top of the jet that quickly breaks down into a large number of smaller vortices as the flow transitions towards turbulence. From the original velocity data, vorticity magnitude is computed and subjected to analysis to identify individual vortices as superlevel set components of high vorticity.

Fig. 5.5 shows the roughly five thousand superlevel set components that exist for level 500 at timestep 2000. Even after topological simplification, the split tree of that timestep still consists of more than $100k$ branches. As explained previously, the image database size only grows proportional to the parameter sampling and not to the feature complexity and quantity (Tab. 5.1). Moreover, grouping components based on split tree branches has the advantage that each group constitutes a local component cluster. Toggling the visibility of these groups therefore supports effective spatial peeling. As even hundreds of images can be composed at interactive framerates, the proposed analysis framework enables analysts to quickly browse through time and update parameters. However, to provide more flexibility, it is necessary to generate image databases for a larger number of component groups and persistence intervals, which significantly increases the databases sizes even further. To summarize, the demonstrated case studies show that image databases are not necessarily small, but seem to grow significantly smaller when moving towards extreme-scale simulations [2].

Dataset	#Cells	#Steps	$\lvert\mathbf{C}\rvert\cdot\lvert\mathbf{L}\rvert\cdot\lvert\mathbf{P}\rvert\cdot\mathbf{n}$	$\mathbf{T_A}$	$\mathbf{T_I}$	$\mathbf{S_S}$	$\mathbf{S_A}$	$\mathbf{S_I}$
VF Run1	$25.1 \cdot 10^3$	100	24×5×1×1	2 m	3 m	90 MB	3 MB	1 GB
VF Run2	$25.1 \cdot 10^3$	100	24×5×1×2	2 m	7 m	90 MB	3 MB	2 GB
VF Run3	$25.1 \cdot 10^3$	100	24×5×1×3	2 m	15 m	90 MB	3 MB	3 GB
yA31	$12.4 \cdot 10^7$	260	24×6×2×4	45 h	43 h	121 GB	28 MB	21 GB
yA31	$2.6 \cdot 10^6$	260	24×6×2×4	16 h	13 h	26 GB	17 MB	20 GB
yB31	$2.6 \cdot 10^6$	260	24×6×2×4	17 h	13 h	26 GB	12 MB	19 GB
yC31	$2.6 \cdot 10^6$	260	24×6×2×4	15 h	14 h	26 GB	14 MB	22 GB
Jet	$3.3 \cdot 10^7$	3000	24×6×2×4	25 h	15 d	375 GB	108 MB	260 GB

Table 5.1: Statistics of the presented case studies. From left to right: dataset name, cell count (all regular grids), number of timesteps, image sampling, total aggregated computation time of analysis and image products, and total size of simulations, analysis products, and images.

5.5 DISCUSSION

This chapter described a scalable processing pipeline that enables the interactive visual analysis of large-scale scientific simulations where superlevel set components and their evolution are of primary interest. The approach first stores analysis products and images during simulation runtime in a Cinema database that can later be used during *post hoc* analysis to efficiently explore the underlying simulation in a topology-based visual analytics framework. To this end, the approach includes a split tree segmentation-based tracking algorithm, and a branch decomposition-based image generation algorithm. The core element of the framework is a dynamic nested tracking graph that illustrates the evolution of components across time and different levels in one compact visualization. A novel graph-based algorithm is capable of deriving NTGs at interactive framerates solely based on the *in situ* generated database. By interacting with this graph, users can query the database in a focus+context approach for other relevant analysis and image elements. Thus, for the first time, the presented methodology enables users to generate and navigate Cinema databases with a focus on features rather than along pre-determined parameter axes. All presented algorithms have been implemented in the *Topology ToolKit* [104] and are accessible as VTK filters [94] inside ParaView [1]. In conclusion, the proposed approach enables users to effectively and efficiently explore simulations containing numerous components.

Regarding future work, the proposed methodology can be improved in several aspects. Importantly, while theoretically feasible, the database generation has not yet been examined during massively parallel *in situ* execution that is needed to allow scaling to state-of-the-art, largest-scale simulations, which stand to benefit from the proposed methodology. This hinges crucially on scalability of the split tree computation that is central to the approach; here, e.g., the *parallel peak pruning* [14] approach could be used. To facilitate practical usability, it appears possible to automate the parameter sampling—e.g., dynamically determining persistence and level intervals—through heuristics or optimization techniques, in order to keep the generated database within a given bandwidth budget while maximizing *post hoc* flexibility, or alternatively, to ensure a specified degree of flexibility while minimizing the database size. For example, the integration of the VOIDGA approach [64] (Ch. 4) in the image generation process will reduce the number of camera samples, which enables a more dense sampling of other parameters. Finally, the database generation could benefit from low-level technical improvements, such as different compression methods and data formats.

CHAPTER 6

CONCLUSION

This work described and evaluated a topology-based methodology that enables the effective *post hoc* analysis of large-scale simulations where the evolution of level, sublevel, and superlevel set components is of primary interest. The backbone of the methodology is a novel topological abstraction, called the nested tracking graph (NTG), that records the evolution of features that exhibit a nesting hierarchy. NTGs are excellent tools to examine the evolution of numerous features inherent in large-scale simulations by partitioning edges into groups based on their nesting hierarchy. In contrast to current state-of-the-art tracking approaches, NTGs simultaneously visualize feature evolution across multiple parameters in one compact representation, effectively setting multiple tracking graphs for individual parameter values in context to each other. Integrated in visual analytic frameworks, NTGs provide an intuitive interaction device that enables analysts to browse through time, filter and select components, and—most importantly—to aggregate components into a manageable amount of groups that can be examined recursively in a level-of-detail approach.

The second half of the manuscript extended their application to large-scale simulations by (i) introducing a novel view-approximation oriented image database generation approach (VOIDGA) that enables the *post hoc* visual exploration of features, and (ii) by describing a *post hoc* tracking algorithm that enables the efficient computation of NTGs for any parameter values without requiring access to the original simulation data. Specifically, instead of explicitly storing feature geometries—which is often infeasible for large-scale simulations due to bandwidth and disk space constraints—the described methodology generates, at simulation runtime, a database consisting of intermediate analysis products—such as depth images, split trees, and meta-edges—that can be used during *post hoc* analysis to compose 3D views, track features, and visualize feature properties. The key advantage of this methodology is that the database only grows proportional to a predefined parameter sampling—e.g., based on a maximum number of images, the resolution of level intervals, or the limit of component groups—independent of the actual size of the simulation data and the number of features. This advantage can not be stressed enough, since the simulation size and the feature complexities are the primary obstacles that need to be overcome to provide flexible, effective, interactive, *post hoc* analysis at the era of exascale computing.

Regarding future research directions, several improvements appear possible. An essential element of the described methodology is the parameter sampling as it predefines the flexibility during *post hoc* analysis, and therefore the choice of the sampling precision has a significant impact on the effectiveness of the approach. VOIDGA reduces one parameter dimension by minimizing the number of camera locations while supporting free camera movement, i.e., instead of storing a fixed set of camera locations, VOIDGA stores a reduced set of images that enables the adequate approximation of any view angle. It is conceivable to derive a similar sampling strategy for other parameters, e.g., by dynamically determining significant persistence and level intervals based on the current simulation state. Moreover, VOIDGA currently generates an image database for each timestep individually, but it appears fruitful to integrate a view interpolation technique across timesteps, which would allow to discard images that can already be interpolated with an acceptable visual error. This will further reduce the size of the database, which frees up space that can be used to sample other parameters more thoroughly.

To improve the accuracy of the *post hoc* tracking, additional topological abstractions besides merge tree segmentations could be used to match features, e.g., persistence pairs [100], Jacobi sets [23], or Morse-Smale complexes [28]. However, these approaches might need to be adapted, as the described methodology requires that the used tracking algorithms satisfy the nesting property (Def. 50) to represent their results via NTGs. In future research, it would be very interesting to examine if this criterion can be relaxed.

BIBLIOGRAPHY

[1] J. Ahrens, B. Geveci, and C. Law. ParaView: An End-User Tool for Large-Data Visualization. *The Visualization Handbook*, pp. 717–731, 2005.

[2] J. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. H. Rogers, and M. Petersen. An Image-Based Approach to Extreme Scale In Situ Visualization and Analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 424–434. IEEE Press, 2014.

[3] G. Aldrich, J. Lukasczyk, J. D. Hyman, G. Srinivasan, H. Viswanathan, C. Garth, H. Leitte, J. Ahrens, and B. Hamann. A Query-Based Framework for Searching, Sorting, and Exploring Data Ensembles. In *IEEE Computing in Science and Engineering*, 2018.

[4] A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Lukić, and E. Van Andel. Nyx: A Massively Parallel AMR Code for Computational Cosmology. *The Astrophysical Journal*, 765(1):39, 2013.

[5] T. F. Banchoff. Critical Points and Curvature for Embedded Polyhedral Surfaces. *The American Mathematical Monthly*, 77(5):475–485, 1970.

[6] U. Bauer, C. Lange, and M. Wardetzky. Optimal Topological Simplification of Discrete Functions on Surfaces. *Discrete & Computational Geometry*, 47(2):347–377, 2012.

[7] J. Bennett, V. Krishnamoorthy, S. Liu, R. W. Grout, E. R. Hawkes, J. H. Chen, J. Shepherd, V. Pascucci, and P.-T. Bremer. Feature-Based Statistical Analysis of Combustion Simulation Data. *IEEE Trans. Vis. Comput. Graph.*, 17(12):1822–1831, 2011.

[8] W. Bethel, B. Tierney, J. Lee, D. Gunter, and S. Lau. Using High-Speed WANs and Network Data Caches to enable Remote and Distributed Visualization. In *Supercomputing, ACM/IEEE 2000 Conference*, pp. 28–28. IEEE, 2000.

[9] T. Biedert and C. Garth. Contour Tree Depth Images for Large Data Visualization. In C. Dachsbacher and P. Navrátil, eds., *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2015.

[10] P.-T. Bremer, E. Bringa, M. Duchaineau, A. Gyulassy, D. Laney, A. Mascarenhas, and V. Pascucci. Topological Feature Extraction and Tracking. In *Journal of Physics: Conference Series*, vol. 78, p. 012007. IOP Publishing, 2007.

[11] P.-T. Bremer, B. Hamann, H. Edelsbrunner, and V. Pascucci. A Topological Hierarchy for Functions on Triangulated Surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):385–396, 2004.

[12] P.-T. Bremer, G. Weber, J. Tierny, V. Pascucci, M. Day, and J. Bell. Interactive Exploration and Analysis of Large-Scale Simulations Using Topology-Based Data Segmentation. *IEEE Transactions on Visualization and Computer Graphics*, 17:1307–1324, 2011.

[13] H. Carr, J. Snoeyink, and U. Axen. Computing Contour Trees in all Dimensions. *Computational Geometry*, 24(2):75 – 94, 2003.

[14] H. Carr, G. Weber, C. M. Sewell, and J. P. Ahrens. Parallel Peak Pruning for Scalable SMP Contour Tree Computation. In *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 75–84. IEEE, 2016.

[15] K. Chen and D. A. Lorenz. Image Sequence Interpolation based on Optical Flow, Segmentation, and Optimal Control. *IEEE Transactions on Image Processing*, 21(3):1020–1030, 2012.

[16] S. E. Chen and L. Williams. View Interpolation for Image Synthesis. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pp. 279–288. ACM, New York, NY, USA, 1993.

[17] M. Ciżnicki, M. Kierzynka, K. Kurowski, B. Ludwiczak, K. Napierała, and J. Palczyński. Efficient Isosurface Extraction using Marching Tetrahedra and Histogram Pyramids on Multiple GPUs. In *International Conference on Parallel Processing and Applied Mathematics*, pp. 343–352. Springer, 2011.

[18] K. Cole-McLaughlin, H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci. Loops in Reeb Graphs of 2-Manifolds. *Discrete & Computational Geometry*, 32(2):231–244, 2004.

[19] J. Cui, Z. Ma, and V. Popescu. Animated Depth Images for Interactive Remote Visualization of Time-Varying Data Sets. *IEEE transactions on visualization and computer graphics*, 20(11):1474–1489, 2014.

[20] A. De Wit, Y. Bertho, and M. Martin. Viscous fingering of miscible slices. *Physics of fluids*, 17:054114, 2005.

[21] P. E. Debevec, C. J. Taylor, and J. Malik. Modeling and Rendering Architecture from Photographs: A Hybrid Geometry-and Image-Based Approach. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 11–20. ACM, 1996.

[22] A. Doi and A. Koide. An Efficient Method of Triangulating Equi-Valued Surfaces by Using Tetrahedral Cells. *IEICE TRANSACTIONS on Information and Systems*, 74(1):214–224, 1991.

[23] H. Edelsbrunner and J. Harer. *Jacobi Sets*, pp. 37–57. London Mathematical Society Lecture Note Series. Cambridge University Press, 2004.

[24] H. Edelsbrunner and J. Harer. Persistent Homology - A Survey. *Contemporary mathematics*, 453:257–282, 2008.

[25] H. Edelsbrunner and J. Harer. *Computational Topology: An Introduction*. American Mathematical Soc., 2010.

[26] H. Edelsbrunner, J. Harer, A. Mascarenhas, and V. Pascucci. Time-varying Reeb Graphs for Continuous Space-time Data. In *Symposium on Computational Geometry*, pp. 366–372. Citeseer, 2004.

[27] H. Edelsbrunner, J. Harer, and A. K. Patel. Reeb Spaces of Piecewise Linear Mappings. In *Symposium on Computational Geometry*, pp. 242–250, 2008.

[28] H. Edelsbrunner, J. Harer, and A. Zomorodian. Hierarchical Morse-Smale Complexes for Piecewise Linear 2-Manifolds. *Discrete and computational Geometry*, 30(1):87–107, 2003.

[29] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological Persistence and Simplification. *Discrete & Computational Geometry*, 28(4):511–533, 2002.

[30] H. Edelsbrunner, D. Morozov, and V. Pascucci. Persistence-Sensitive Simplification Functions on 2-Manifolds. In *Proceedings of the twenty-second annual symposium on Computational geometry*, pp. 127–134. ACM, 2006.

[31] H. Edelsbrunner and E. P. Mücke. Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms. *ACM Transactions on Graphics (tog)*, 9(1):66–104, 1990.

[32] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull. Graphviz—Open Source Graph Drawing Tools. In *International Symposium on Graph Drawing*, pp. 483–484. Springer, 2001.

[33] J. Feder. Viscous Fingering in Porous Media. In *Fractals*, pp. 41–61. Springer, 1988.

[34] O. Fernandes, S. Frey, F. Sadlo, and T. Ertl. Space-Time Volumetric Depth Images for In-Situ Visualization. In *IEEE 4th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 59–65, 2014.

[35] M. Gittings, R. Weaver, M. Clover, T. Betlach, N. Byrne, R. Coker, E. Dendy, R. Hueckstaedt, K. New, W. R. Oakes, et al. The RAGE Radiation-Hydrodynamic Code. *Computational Science & Discovery*, 1(1):015005, 2008.

[36] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny. Task-Based Augmented Merge Trees with Fibonacci Heaps. In *IEEE Symposium on Large Data Analysis and Visualization 2017*, 2017.

[37] T. Gurdan, M. R. Oswald, D. Gurdan, and D. Cremers. Spatial and temporal interpolation of multi-view image sequences. In *German Conference on Pattern Recognition*, pp. 305–316. Springer, 2014.

[38] A. Gyulassy, V. Natarajan, V. Pascucci, and B. Hamann. Efficient Computation of Morse-Smale Complexes for Three-Dimensional Scalar Functions. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1440–1447, 2007.

[39] C. Hane, L. Ladicky, and M. Pollefeys. Direction matters: Depth estimation with a surface normal classifier. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 381–389, 2015.

[40] C. Heine, H. Leitte, M. Hlawitschka, F. Iuricich, L. De Floriani, G. Scheuermann, H. Hagen, and C. Garth. A Survey of Topology-Based Methods in Visualization. In *Computer Graphics Forum*, vol. 35, pp. 643–667. Wiley Online Library, 2016.

[41] J. Hocking and G. Young. Topology. *Addison Wesley*, 1961.

[42] IEEEVIS. Scientific Visualization Contest 2016. http://www.uni-kl.de/sciviscontest/, 2016.

[43] IEEEVIS. Scientific Visualization Contest 2018. http://sciviscontest2018.org/, 2018.

[44] G. Ji and H.-W. Shen. Feature tracking using earth mover's distance and global optimization. In *Pacific Graphics*, vol. 2, 2006.

[45] G. Ji, H.-W. Shen, and R. Wenger. Volume Tracking using Higher Dimensional Isosurfacing. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, p. 28. IEEE Computer Society, 2003.

[46] L. V. Kantorovich. On the Translocation of Masses. *Journal of Mathematical Sciences*, 133(4):1381–1382, 2006.

[47] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[48] W. Köpp and T. Weinkauf. Temporal Treemaps: Static Visualization of Evolving Trees. *IEEE transactions on visualization and computer graphics*, 25(1):534–543, 2019.

[49] M. Krivokuća, B. C. Wünsche, and W. Abdulla. A new Error Metric for Geometric Shape Distortion using Depth Values from Orthographic Projections. In *Proceedings of the 27th Conference on Image and Vision Computing New Zealand*, pp. 388–393. ACM, 2012.

[50] P. Lacroute and M. Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pp. 451–458. ACM, New York, NY, USA, 1994.

[51] H. G. Lalgudi, M. W. Marcellin, A. Bilgin, H. Oh, and M. S. Nadar. View Compensated Compression of Volume Rendered Images for Remote Visualization. *IEEE Transactions on Image Processing*, 18(7):1501–1511, 2009.

[52] A. G. Landge, V. Pascucci, A. Gyulassy, J. C. Bennett, H. Kolla, J. Chen, and P.-T. Bremer. In-Situ Feature Extraction of Large Scale Combustion Simulations using Segmented Merge Trees. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pp. 1020–1031. IEEE, 2014.

[53] D. Laney, A. Mascarenhas, P. Miller, V. Pascucci, et al. Understanding the Structure of the Turbulent Mixing Layer in Hydrodynamic Instabilities. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1053–1060, 2006.

[54] M. Levoy and P. Hanrahan. Light Field Rendering. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pp. 31–42. ACM, New York, NY, USA, 1996.

[55] B. Li, C. Shen, Y. Dai, A. van den Hengel, and M. He. Depth and surface normal estimation from monocular images using regression on deep features and hierarchical CRFs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1119–1127, 2015.

[56] J. Li, R. Klein, and A. Yao. A Two-Streamed Network for Estimating Fine-Scaled Depth Maps from Single RGB Images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3372–3380, 2017.

[57] C. Lipski, C. Linz, K. Berger, A. Sellent, and M. Magnor. Virtual Video Camera: Image-Based Viewpoint Navigation through Space and Time. In *Computer Graphics Forum*, vol. 29, pp. 2555–2568. Wiley Online Library, 2010.

[58] F. Liu, C. Shen, G. Lin, and I. Reid. Learning Depth from Single Monocular Images using Deep Convolutional Neural Fields. *IEEE transactions on pattern analysis and machine intelligence*, 38(10):2024–2039, 2016.

[59] W. E. Lorensen and H. E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *ACM siggraph computer graphics*, vol. 21, pp. 163–169. ACM, 1987.

[60] J. Lukasczyk. Cinema modules of the Topology ToolKit (TTK). https://topology-tool-kit.github.io/, 2019.

[61] J. Lukasczyk. Nested Tracking Graph modules of the Topology ToolKit (TTK). https://topology-tool-kit.github.io/, 2019.

[62] J. Lukasczyk. VOIDGA modules of the Topology ToolKit (TTK). https://topology-tool-kit.github.io/, 2019.

[63] J. Lukasczyk, G. Aldrich, M. Steptoe, G. Favelier, C. Gueunet, J. Tierny, R. Maciejewski, B. Hamann, and H. Leitte. Viscous Fingering: A Topological Visual Analytic Approach. In *Applied Mechanics and Materials*, vol. 869, pp. 9–19. Trans Tech Publ, 2017.

[64] J. Lukasczyk, E. Kinner, J. Ahrens, H. Leitte, and C. Garth. VOIDGA: A View-Approximation Oriented Image Database Generation Approach. In *IEEE 8th*

Symposium on Large Data Analysis and Visualization (LDAV) [Best Paper Award], 2018.

[65] J. Lukasczyk, R. Maciejewski, C. Garth, and H. Hagen. Understanding Hotspots: A Topological Visual Analytics Approach. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '15, pp. 36:1–36:10. ACM, 2015.

[66] J. Lukasczyk, G. Weber, R. Maciejewski, C. Garth, and H. Leitte. Nested Tracking Graphs. In *Computer Graphics Forum [Best Paper Award]*, vol. 36, pp. 12–22, 2017.

[67] Z. Lukić, C. W. Stark, P. Nugent, M. White, A. A. Meiksin, and A. Almgren. The Lyman α Forest in Optically Thin Hydrodynamical Simulations. *Monthly Notices of the Royal Astronomical Society*, 446(4):3697–3724, 2014.

[68] K.-L. Ma. In Situ Visualization at Extreme Scale: Challenges and Opportunities. *IEEE Computer Graphics and Applications*, 29(6):14–19, 2009.

[69] S. Maadasamy, H. Doraiswamy, and V. Natarajan. A Hybrid Parallel Algorithm for Computing and Tracking Level Set Topology. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pp. 1–10. IEEE, 2012.

[70] A. Mascarenhas and J. Snoeyink. *Isocontour based Visualization of Time-varying Scalar Fields*, pp. 41–68. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[71] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pp. 39–46. ACM, 1995.

[72] J. W. Milnor, M. Spivak, R. Wells, and R. Wells. *Morse Theory*. Princeton university press, 1963.

[73] D. Morozov and G. Weber. Distributed Merge Trees. In *Acm sigplan notices*, vol. 48, pp. 93–102. ACM, 2013.

[74] D. Morozov and G. Weber. Distributed Contour Trees. In *Topological Methods in Data Analysis and Visualization III*, pp. 89–102. Springer, 2014.

[75] K. Mueller, A. Smolic, K. Dix, P. Merkle, P. Kauff, and T. Wiegand. View Synthesis for Advanced 3D Video Systems. *EURASIP Journal on image and video processing*, 2008(1):438148, 2009.

[76] NASA. Planetary Defense Coordination Office. https://www.nasa.gov/planetarydefense, 2018.

[77] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon. KinectFusion: Real-Time Dense Surface Mapping and Tracking. In *Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on*, pp. 127–136. IEEE, 2011.

[78] Z. Ni, D. Tian, S. Bhagavathy, J. Llach, and B. S. Manjunath. Improving the Quality of Depth Image Based Rendering for 3D Video Systems. In *Image Processing (ICIP), 2009 16th IEEE International Conference on*, pp. 513–516. IEEE, 2009.

[79] P. Oesterling, C. Heine, G. Weber, D. Morozov, and G. Scheuermann. Computing and Visualizing Time-Varying Merge Trees for High-Dimensional Data. In *Topological Methods in Data Analysis and Visualization*, pp. 87–101. Springer, 2015.

[80] J. Ogniewski. High-Quality Real-Time Depth-Image-Based-Rendering. In *Proceedings of SIGRAD 2017, August 17-18, 2017 Norrköping, Sweden*, pp. 1–8. Linköping University Electronic Press, 2017.

[81] M. M. Oliveira. Image-based modeling and rendering techniques: A survey. *RITA*, 9(2):37–66, 2002.

[82] Open-Source Cinema Viewers. CVLIB. http://cinemaviewer.org/, 2018.

[83] P. O'Leary, J. Ahrens, S. Jourdain, S. Wittenburg, D. H. Rogers, and M. Petersen. Cinema Image-Based In Situ Analysis and Visualization of MPAS-Ocean Simulations. *Parallel Computing*, 55:43–48, 2016.

[84] V. Pascucci, K. Cole-McLaughlin, and G. Scorzelli. Multi-Resolution Computation and Presentation of Contour Trees. In *Proc. IASTED Conference on Visualization, Imaging, and Image Processing*, pp. 452–290. Citeseer, 2004.

[85] J. Patchett, G. Gisler, B. Nouanesengsy, D. H. Rogers, G. Abram, F. Samsel, K. Tsai, and T. Turton. Visualization and Analysis of Threats from Asteroid Ocean Impacts. *Los Alamos National Laboratory*, 2016.

[86] F. H. Post, B. Vrolijk, H. Hauser, R. S. Laramee, and H. Doleisch. The State of the Art in Flow Visualisation: Feature Extraction and Tracking. In *Computer Graphics Forum*, vol. 22, pp. 775–792. Wiley Online Library, 2003.

[87] G. Reeb. Sur les points singuliers dune forme de Pfaff compltement intgrable ou dune fonction numrique. *Comptes Rendus des séances de lAcadémie des sciences*, 222(847-849):76, 1946.

[88] E. Reinhard, W. Heidrich, P. Debevec, S. Pattanaik, G. Ward, and K. Myszkowski. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. Morgan Kaufmann, 2010.

[89] B. Rieck, U. Fugacci, J. Lukasczyk, and H. Leitte. Clique Community Persistence: A Topological Visual Analysis Approach for Complex Networks. *IEEE transactions on visualization and computer graphics*, 24(1):822–831, 2018.

[90] D. Rogers, J. Woodring, J. Ahrens, J. Patchett, and J. Lukasczyk. Cinema Database Specification - Dietrich Release v1.2. Technical Report LA-UR-17-25072, Los Alamos National Laboratory, 2018.

[91] H. Saikia and T. Weinkauf. Global Feature Tracking and Similarity Estimation in Time-Dependent Scalar Fields. In *Computer Graphics Forum*, vol. 36, pp. 1–11. Wiley Online Library, 2017.

[92] R. Samtaney, D. Silver, N. Zabusky, and J. Cao. Visualizing Features and Tracking their Evolution. *Computer*, 27(7):20–27, 1994.

[93] W. Schroeder, R. Maynard, and B. Geveci. Flying Edges: A High-Performance Scalable Isocontouring Algorithm. In *2015 IEEE 5th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 33–40. IEEE, 2015.

[94] W. J. Schroeder, K. Martin, and W. E. Lorensen. *The Visualization Toolkit: An Object Oriented Approach to 3D Graphics (3rd edition)*. Kitware, Inc., 2004.

[95] H. Shum and S. B. Kang. Review of Image-Based Rendering Techniques. In *Visual Communications and Image Processing 2000*, vol. 4067, pp. 2–14. International Society for Optics and Photonics, 2000.

[96] D. Silver and X. Wang. Tracking and Visualizing Turbulent 3D Features. *IEEE Transactions on Visualization and Computer Graphics*, 3:129–141, 1997.

[97] D. Silver and X. Wang. Tracking Scalar Features in Unstructured Data Sets. In *Proceedings Visualization'98 (Cat. No. 98CB36276)*, pp. 79–86. IEEE, 1998.

[98] A. Smolic, K. Muller, K. Dix, P. Merkle, P. Kauff, and T. Wiegand. Intermediate View Interpolation based on Multiview Video Plus Depth for Advanced 3D

Video Systems. In *Image Processing, 2008. ICIP 2008. 15th IEEE International Conference on*, pp. 2448–2451. IEEE, 2008.

[99] B.-S. Sohn and C. Bajaj. Time-Varying Contour Topology. *IEEE Transactions on Visualization and Computer Graphics*, 12:14–25, 2006.

[100] M. Soler, M. Plainchault, B. Conche, and J. Tierny. Lifted Wasserstein Matcher for Fast and Robust Topology Tracking. In *IEEE 8th Symposium on Large Data Analysis and Visualization (LDAV)*, 2018.

[101] T. Stich, C. Linz, C. Wallraven, D. Cunningham, and M. Magnor. Perception-Motivated Interpolation of Image Sequences. *ACM Transactions on Applied Perception (TAP)*, 8(2):11, 2011.

[102] W. Sun, L. Xu, O. C. Au, S. H. Chui, and C. W. Kwok. An Overview of Free View-Point Depth-Image-Based Rendering (DIBR). In *APSIPA Annual Summit and Conference*, pp. 1023–1030, 2010.

[103] J. Tierny. *Contributions to Topological Data Analysis for Scientific Visualization*. Habilitation thesis, UPMC-Paris 6 Sorbonne Universités, 2016.

[104] J. Tierny, G. Favelier, J. A. Levine, C. Gueunet, and M. Michaux. The Topology ToolKit. *IEEE Transactions on Visualization and Computer Graphics*, 2017.

[105] J. Tierny, A. Gyulassy, E. Simon, and V. Pascucci. Loop Surgery for Volumetric Meshes: Reeb Graphs Reduced to Contour Trees. *IEEE Transactions on Visualization and Computer Graphics*, 15(6), 2009.

[106] J. Tierny and V. Pascucci. Generalized Topological Simplification of Scalar Fields on Surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2005–2013, 2012.

[107] J. Unger, A. Gardner, P. Larsson, and F. Banterle. Capturing Reality for Computer Graphics Applications. In *SIGGRAPH Asia 2015 Courses*, p. 7. ACM, 2015.

[108] M. Van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour Trees and Small Seed Sets for Isosurface Traversal. In *Proceedings of the thirteenth annual symposium on Computational geometry*, pp. 212–220. ACM, 1997.

[109] Q. Wang, J. JaJa, and A. Varshney. An Efficient and Scalable Parallel Algorithm for Out-Of-Core Isosurface Extraction and Rendering. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 10–pp. IEEE, 2006.

[110] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image Quality Assessment: from Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[111] Z. Wang, E. P. Simoncelli, and A. C. Bovik. Multiscale Structural Similarity for Image Quality Assessment. In *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Seventh Asilomar Conference on*, vol. 2, pp. 1398–1402. Ieee, 2003.

[112] G. Weber, P.-T. Bremer, M. Day, J. Bell, and V. Pascucci. Feature Tracking using Reeb Graphs. In *Topological Methods in Data Analysis and Visualization*, pp. 241–253. Springer, 2011.

[113] M. Werlberger, T. Pock, M. Unger, and H. Bischof. Optical flow guided TV-L1 video interpolation and restoration. In *International Workshop on Energy Minimization Methods in Computer Vision and Pattern Recognition*, pp. 273–286. Springer, 2011.

[114] W. Widanagamaachchi, C. Christensen, V. Pascucci, and P.-T. Bremer. Interactive Exploration of Large-Scale Time-Varying Data using Dynamic Tracking Graphs. In *Large data analysis and visualization (LDAV), 2012 IEEE Symposium on*, pp. 9–17. IEEE, 2012.

[115] J. Woodring, J. P. Ahrens, J. Patchett, C. Tauxe, and D. H. Rogers. High-Dimensional Scientific Data Exploration via Cinema. In *2017 IEEE Workshop on Data Systems for Interactive Analysis (DSIA)*, pp. 1–5. IEEE, 2017.

[116] H. Xu and B. Chen. Stylized Rendering of 3D Scanned Real World Environments. In *Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*, pp. 25–34. ACM, 2004.

[117] H. M. Yilmaz, M. Yakar, S. A. Gulec, and O. N. Dulgerler. Importance of Digital Close-Range Photogrammetry in Documentation of Cultural Heritage. *Journal of Cultural Heritage*, 8(4):428–433, 2007.

[118] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K.-L. Ma. In Situ Visualization for Large-Scale Combustion Simulations. *IEEE computer graphics and applications*, 30(3):45–57, 2010.

[119] F. Zhang, S. Lasluisa, T. Jin, I. Rodero, H. Bui, and M. Parashar. In-situ feature-based objects tracking for large-scale scientific simulations. In *2012 SC Com-*

panion: High Performance Computing, Networking Storage and Analysis, pp. 736–740. IEEE, 2012.

[120] S. Zinger, L. Do, and P. H. N. de With. Free-Viewpoint Depth Image Based Rendering. *Journal of visual communication and image representation*, 21(5-6):533–541, 2010.

[121] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface Splatting. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pp. 371–378. ACM, New York, NY, USA, 2001.

CURRICULUM VITAE

Jonas Lukasczyk

Education

01/2015 - 07/2019 **Dr. rer. nat. in Computer Science**
Technische Universität Kaiserslautern, Germany

Member of IRTG 2057 (http://www.irtg2057.de/)

Research Topic:

- Topology-Based Characterization and Visual Analysis of Feature Evolution in Large-Scale Simulations

09/2012 - 01/2015 **M. Sc. in Applied Computer Science**
Technische Universität Kaiserslautern, Germany

Master Thesis:

- Visualization and Analysis of Spatio-Temporal Trends

Focus Areas:

- Computational Geometry and Topology
- Scientific Visualization
- Mathematical Modelling

09/2009 - 09/2012 **B. Sc. in Applied Computer Science**
Technische Universität Kaiserslautern, Germany

Bachelor Thesis:

- Probabilistic Execution Time Estimation of Sequentially Executed Embedded Systems

Experience

12/2017 - 07/2019 Research Associate, Visual Information Analysis Group, TU Kaiserslautern

10/2014 - 12/2017 Research Assistant, Scientific Visualization Lab, TU Kaiserslautern

08/2013 - 09/2014 Research Assistant, Image Processing Department, Fraunhofer ITWM

05/2011 - 07/2013 Research Assistant, Safety and Reliability Department, Fraunhofer IESE

List of Publications

- J. Lukasczyk, C. Garth, T. Biedert, R. Maciejewski, G.H. Weber, H. Leitte. *Dynamic Nested Tracking Graphs*. EEE Transactions on Visualization and Computer Graphics. 2019.

- G. Aldrich, J. Lukasczyk, J.D. Hyman, G. Srinivasan, H. Viswanathan, C. Garth, H. Leitte, J. Ahrens, and B. Hamann. *A Query-based Framework for Searching, Sorting, and Exploring Data Ensembles.* Computing in Science and Engineering. 2019.

- A. Middel, J. Lukasczyk, S. Zakrzewski, M. Arnold, and R. Maciejewski. *Urban Form and Composition of Street Canyons: A Human-Centric Big Data and Deep Learning Approach*. Landscape and Urban Planning 183: 122-132. 2019.

- J. Lukasczyk, E. Kinner, J. Ahrens, H. Leitte, and C. Garth. *VOIDGA: A View-Approximation Oriented Image Database Generation Approach [Best Paper Award]*. Proceedings of IEEE Symposium on Large Data Analysis and Visualization (LDAV). 2018.

- A. Middel, J. Lukasczyk, R. Maciejewski, M. Demuzere, and M. Roth. *Sky View Factor Footprints for Urban Climate Modeling*. Urban Climate, Volume 25, Pages 120-134. 2018.

- C. Wang, A. Middel, S.W. Myint, S. Kaplan, A.J. Brazel, and J. Lukasczyk. *Assessing Local Climate Zones in Arid Cities: The Case of Phoenix, Arizona and Las Vegas, Nevada*. ISPRS Journal of Photogrammetry and Remote Sensing, Volume 141, Pages 59-71. 2018.

- B. Rieck, U. Fugacci, J. Lukasczyk, and H. Leitte. *Clique Community Persistence: A Topological Visual Analysis Approach for Complex Networks*. IEEE Transactions on Visualization and Computer Graphics. 2017.

- J. Lukasczyk, G.H. Weber, R. Maciejewski, C. Garth, and H. Leitte. *Nested Tracking Graphs [Best Paper Award]*. Computer Graphics Forum (Special Issue, Proceedings Eurographics/IEEE Symposium on Visualization). Vol. 36. No. 3. 2017.

- J. Lukasczyk, G. Aldrich, M. Steptoe, G. Favelier, C. Gueunet, J. Tierny, R. Maciejewski, B. Hamann, and H. Leitte. *Viscous Fingering: A Topological Visual Analytic Approach*. Applied Mechanics and Materials 869 - Proceedings of the 1st Conference on Physical Modeling for Virtual Manufacturing Systems and Processes (2017): S. 9-19. 2017.

- A. Middel, J. Lukasczyk, and R. Maciejewski. *Sky View Factors from Synthetic Fisheye Photos for Thermal Comfort Routing - A Case Study in Phoenix, Arizona*. Urban Planning 2.1. 2017.

- G. Aldrich, J. Lukasczyk, M. Steptoe, R. Maciejewski, H. Leitte, and B. Hamann. *Viscous Fingers: A Topological Visual Analytics Approach [Honorable Mention]*. IEEE Vis Contest. 2016.
- J. Lukasczyk, R. Maciejewski, C. Garth, and H. Hagen. *Understanding Hotspots: A Topological Visual Analytics Approach*. ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. 2015.
- J. Lukasczyk, X. Liang, W. Luo, E.D. Ragan, A. Middel, N. Bliss, D. White, H. Hagen, and R. Maciejewski. *A Collaborative Web-Based Environmental Data Visualization and Analysis Framework*. In Proc. Workshop on Visualization and Environmental Sciences (EnvirVis). 2015.
- J. Lukasczyk, A. Middel, and H. Hagen. *WebGL-Based Geodata Visualization for Policy Support and Decision Making*. In Proc. Workshop on Visualization and Environmental Sciences (EnvirVis). 2014.