Erklärung:

Hiermit erkäre ich gemäß §18 Abs. 9 der Diplomprüfungsordnung, daß ich diese Arbeit selbstständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kaiserslautern, den 13. Mai 1999

(Gunther. H. Weber)

Diplomarbeit

Interactive Visualization of Vector Fields Using Tetrahedral Hierarchies

Gunther H. Weber

AG Graphische Datenverarbeitung und Computergeometrie
Fachbereich Informatik
Universität Kaiserslautern

May 11, 1999

to my mother...

And so it begins. You have forgotten something.

(Kosh — Babylon 5)

This Diplomarbeit would not have been possible, if it were not for several people, whose support made it possible. I want to use this opportunity to thank them.

The collaboration with Prof. Dr. Bernd Hamann[1], Prof. Dr. Hans Hagen and Prof. Dr. Ken Joy[1] spawned this work. Their support, experience and encouragement helped me to achieve what I did.

Björn Heckel's insight in clustering techniques was of valuable help during my work on the first part of this Diplomarbeit. Oliver Kreylos implemented serveral demonstration programs on the Immersive WorkBench. Looking at his examples simplified the implementation of the visualization environment considerably. His designs were building blocks in the actual implementation. Jörg Meyer and Gerik Scheuermann helped me in the proof reading process for the documentation.

I want to thank my parents for providing for my education, leading me through the first part of my life and contributing to me becoming the person I am now.

Additionally I would like to thank my friends for their support before and during this Diplomarbeit and sometimes taking my mind of my work.

Last but not least I want to use this opportunity to thank my former teacher Hans-Karl Keller. During my time at the Gauß Gymnasium he supported my interest in computer science by giving me the opportunity to work in the "Pascal Grafik AG" and later in the "Informatik AG" and spawned my interest in the formal and theoretical aspects of computer science.

[1]Center for Image Processing and Integrated Computing, Dept. of Computer Science, University of California at Davis

Contents

List of Figures

LIST OF FIGURES

Preface

Current numerical simulation and data acquisition techniques have been refined to a point where they are a source of highly accurate descriptions of natural or simulated phenomena and processes. The amount of data collected even by a single satellite in Earth orbit is massive. Unfortunately, a large part of such data can only be stored for further investigation, since data analysis and visualization techniques capable of handling vast amounts of data are yet to be developed.

While the essential visualization techniques are still in the initial stages of development, further steps in the data acquisition process are undertaken, widening the gap between the capacity to produce data and the ability to analyze and visualize them. The area of large scale data visualization is likely to remain a challenge for a long time to go.

Past research in the field of large-scale data visualization mainly focused on scalar data. In this case, for any given point in a volume a scalar value (like density, temperature, etc.) is associated with this point. Large-scale scalar fields can be found in applications like medical imaging where modern *Computerized Axial Tomography* (CAT) and *Magnetic Resonance Imaging* (MRI) scans can contain up to 512^3 scalar values on a rectilinear grid or even more data. Data sets derived from *Computational Fluid Dynamics* (CFD) even have sizes in the gigabyte and terabyte range.

However, there are also many vector fields that are of interest, *e.g.*, data from CFD simulations, measured wind velocity above the Earth and wind tunnel data. It is desirable to use classical vector field visualization techniques like stream lines, stream ribbons and stream surfaces on these large-scale data sets. Because of the sheer size of the data sets it is prohibitive to apply these algorithms to the original data. This necessitates finding an alternate representation for vector fields. The representation should be sufficiently compact to allow interactive visualization of the underlying vector field, but retain enough accuracy to find interesting features of the vector field and examine them in detail.

This Diplomarbeit develops an approach for the visualization of large-scale vector fields. First, a hierarchical representation of the vector field is computed using *clustering*, a classical data analysis technique. This step yields a representation of the vector field, describing it at different levels of detail. Furthermore, the chosen representation allows almost seamless blending between the various levels of detail. Utilizing this representation it is possible to interactively visualize the vector field. In order to interpolate values for a given point of the field, two methods are developed. One method is a localized variant of Hardy's multiquadric. To accommodate visualization techniques relying on a grid, a procedural triangulation scheme, generating the needed grid on the fly, is introduced. After the user has specified a region of interest, the appropriate level of detail is accessed and a localized triangulation of that region is computed and

used for subsequent visualization.

These techniques are used to interactively visualize vector fields on workstations and the *Responsive Workbench*, a *virtual reality* (VR) device developed by the Gesellschaft für Mathematik und Datenverarbeitung (GMD), Bonn, Germany, and Stanford University, California, USA, now mass-produced by Fakespace Inc., Mountain View, California, USA, and sold under the product name *Immersive WorkBench.*

Chapter 1

Introduction

1.1 Vector fields

Vector fields play an important role in physics and engineering. Many physical properties are not only characterized by a scalar quantity like temperature but also by a direction. Vector quantities like velocity describe a magnitude and a direction.

In many cases, a physical property varies in a given spatial region. The temperature in a room varies at different positions within that room. A *scalar field* is a function $F(\mathbf{x})$ that assigns a scalar value to each position x within a given region. Likewise, a *vector field* is a function that assigns a vector value to each position in space.

A common example for a vector field in engineering is measured data from wind tunnel experiments. At each point in the wind tunnel the air has a specific velocity. This velocity consists of two parts. A magnitude giving the speed of the air and the direction in which the air moves in that particular point.

Looking at the raw data as a list of positions and associated vectors does not provide insight into the nature of the vector field. *E.g.*, given a discrete vector field from a wind tunnel experiment, the person looking at the data is interested in the path that a particle would follow if it is injected into the field. *Stream lines* describe the path that this particle will follow. Other vector field visualization techniques simulate experiments that where performed in wind tunnels prior to the availability of computers.

In conjunction with simulation techniques from CFD, it is possible to simulate and evaluate the behavior of the air flow in vicinity of a car or an airplane and detect critical points. It is obvious that the availability of powerful visualization techniques is essential in engineering applications. Using the information gathered by visualizing the air flow, the design can be altered to achieve desired properties (*e.g.*, minimizing drag or maximizing lift for an airplane wing).

In physics vector fields arise in a greater variety. *E.g.*, similar to mechanical engineering one can be interested in the velocity of air or water in a given region. But there is also a great variety of force fields. These fields describe forces that act on particles with given properties (*e.g.*, having a determined electrical charge). *E.g.*, the propagation of electro-magnetic waves is governed by the underlying vector fields.

However, there are applications outside of physics and engineering. Collected data describing wind speed over a given region of the Earth can be visualized to gain new insights into environmental processes and weather changes.

1.2 Large-scale data sets

Large scale data sets arise in a multitude of applications. For example, advances in MRI and CAT technology produce volumetric scalar data at ever increasing resolutions. Recently, techniques to generate vector field data have improved considerably as well. Simulation techniques can generate CFD data with sizes of several terabytes. Likewise, the methods of collecting measured data have improved to a point where the resulting data sets are of similar size.

There are high-resolution data sets of measured wind data over California and the Pacific Ocean that are several terabytes in size. In order to get an overview of an entire data set it is impossible to use the raw data, since it will not fit into the RAM of current workstations. Even loading these data sets takes several minutes or even hours. Accessing the data during the visualization is too slow to be feasible. Additionally, the computational cost of the visualization techniques increases with the data set size. In order to allow visualization at interactive rates (frame rates of 30 frames per second), the amount of data to be processed must be reduced.

When reducing the amount of data of the original data set, it is desirable to find an approximation of the data that is sufficiently small to allow visualizing the data set with standard techniques in a reasonable amount of time. However, the representation must be sufficiently accurate to allow the user to navigate within the vector field and visualize details.

1.3 Requirements leading to a hierarchical approach

Lets assume a given vector field has been reduced for visualization on a workstation. The approximation is sufficiently small to fit into the memory of the used workstation, and existing visualization techniques are able to achieve interactive frame rates on this data set. Thus, the user can get an overview of the properties of the vector field.

If the user finds a part of the vector field that attracts her/his attention she/he needs to be able to examine this region more thoroughly. In order to do this she/he may zoom into that particular region of space, focusing her/his complete attention onto it.

If only an approximation of the vector field over the whole region where it is defined has been computed, this approximation might not be sufficiently accurate, when the user zooms into a certain region of the vector field. Details that are small when looking at the complete vector field and that could be be ignored in a display of the entire vector field may now be needed to understand the vector field in this given region.

In order to do this, additional information is required that refines the representation of the vector field in a particular region. If the user zooms further into the vector field, more information is required to display the vector field in a way such that the difference between the original data set and the approximation will be barely visible.

Since the refinement takes place in a limited region of the vector field, *i.e.*, the currently visible region of the vector field, it is possible to use a more accurate representation without exceeding the amount of data that can be stored in the memory of the workstation.

These requirements make it necessary that the vector field is not stored in a single fixed approximation with a given error bound, but rather in a way enabling the software to refine the data in any given region to a given error bound. In the optimal case, it would be possible to refine the representation until the displayed region is sufficiently small enough to use the original data set for computation. This objective can be achieved by storing the vector field in a *hierarchical representation*. A hierarchical representation stores data at multiple levels of detail, and the relation of these levels allows direct access to the desired level of detail for a given region.

1.4 Hierarchy generation in a pre-processing step

There are several ways to construct hierarchical approximations. *Wavelets* (see [41, 42]) represent a data set by a coarse approximation and information adding increasing levels of detail (high frequency information). In this work, a clustering technique is used to generate approximation levels of the vector field. Clustering is a technique widely used in statistics and collects objects of similar properties in groups called "clusters". This is used to group coherent regions of vectors with similar direction and length into clusters. Each cluster has a *cluster representant* or *cluster center* which is computed by averaging the positions and vectors in the cluster. These representants are subsequently used to represent the data set.

The approximation is generated using a top-down strategy. Initially all samples (position and vector value at that position) are assigned to one cluster. Using a distance measure defined for the samples, in each following step the cluster with the largest deviation of samples from the representant is chosen and divided into two new clusters. This split process is repeated until an approximation of the desired accuracy is generated. By not only storing the representants of the final cluster approximation, but also all intermediate cluster representants and their relation, *i.e.*, at what time a cluster is split and what clusters result from the split process, it is possible to get a hierarchical representation of the vector field. This can be used to seamlessly generate representations of a desired accuracy within given bounds.

Many current visualization techniques require a grid-based representation of a data set, *e.g.*, a space decomposition into tetrahedral cells, which provide connectivity information between the individual positions in a data set. Instead of pre-computing the grids for the complete vector field, a different approach is taken in the course of this Diplomarbeit. If the number of data is not too large, it is possible to generate a triangulation procedurally during the actual visualization step. When the user chooses a region of interest, a representation of the vector field consisting of a fixed number of vectors is generated using the hierarchy resulting from the clustering process. The samples of this representation are used as input for a triangulation scheme (currently *Delaunay triangulation*). Since a finer representation is only used when a user needs to focus on a given region, this approach has the advantage of avoiding the need to compute the triangulation for a complete data set for this level of detail. Only the actually needed part of this

triangulation is generated saving computation time. Furthermore, this approach is very flexible, because it allows all intermediate cluster representations to be used and not only a subset for which a triangulation has been computed. This allows a seamless navigation in the hierarchy.

In some cases, where no connectivity information is needed, a localized version of the *Hardy interpolation* (see [12, 13]) proves to be an alternative to a grid-based interpolation scheme, providing often superior results. Since Hardy interpolation is a grid-less interpolation technique for scattered data and the computed interpolant only considers a fixed number of points in the vicinity of the current query point, the computational cost of this method is small compared to the procedural triangulation approach.

1.5 Visualization

The application of a hierarchical representation is to enable a user to visualize data at different levels of accuracy and gain insight in its properties. Several visualization techniques mimic experiments formerly performed in wind tunnels to analyze a flow, *i.e.*, a vector field of velocities of gas or fluid particles. *Path lines* (or *trajectories*) trace the path of a particle injected into a flow at a given time and position. *Stream lines* remove the time dependency and trace the path a particle would follow, if the vector field was frozen at a given moment in time. In the case of time-independent vector fields path lines and stream lines coincide.

Stream lines can be used as building blocks for additional visualization techniques. *Stream ribbons* result from tracing two adjacent stream lines and connecting the resulting paths with edges, thus defining a polygonal mesh. *Stream surfaces* result from tracing all stream lines that start on a given curve segment. Since this means that an infinite number of stream lines must be traced, stream surfaces are approximated by tracing a finite, sufficiently large number of stream lines and connecting them with a polygonal mesh. Stream surfaces are a powerful tool to visualize properties like *divergence* and *vorticity* of a vector field.

In the visualization part of this Diplomarbeit, the procedural triangulation algorithm, combined with a mesh-based interpolation scheme, and alternatively Hardy Interpolation are used to compute stream lines, stream surfaces, and stream ribbons. These are displayed on a workstation monitor using *OpenInventor.* Additionally, a prototype of a visualization system on the Responsive Workbench is discussed, using the concept of virtual environments to provide an intuitive visualization system allowing to gain an insight into the properties of a vector field that cannot be attained using conventional two-dimensional images on a monitor. Furthermore, interaction techniques with the virtual environment are presented. These enable a user to modify a virtual environment.

Chapter 2

Fundamentals and related work

2.1 Vector fields

Vector fields describe how a vector quantity varies in a given region of space, more formally defined by

Definition 1 (vector field)
Given a region R in space (in most applications an open subset of $\mathbb{R}^n$) and a vector space V (mostly $\mathbb{R}^m$) a vector field is a function $F : R \to V$ that maps a given location $\mathbf{x}$ to a vector value $F(\mathbf{x}) = \mathbf{v}$.

In most practical applications, n equals m, and n is usually two or three (although the combination $n = 2$, $m = 3$ can be found in some applications).

An example for a three-dimensional (3D) vector field is the Coulomb field. A particle with an electrical charge Q at position $\mathbf{p}_0$ exerts an electrostatic force on other electrically charged particles. The force acting on a particle with unit electric charge is given by the analytical function

$$F(\mathbf{p}) = \frac{1}{\varepsilon_0 \varepsilon_r} \frac{Q}{\|\mathbf{p}_0 - \mathbf{p}\|^2} \frac{\mathbf{p}_0 - \mathbf{p}}{\|\mathbf{p}_0 - \mathbf{p}\|} \tag{2.1}$$

with $\|\cdot\|$ being the Euclidean norm and thus $\|\mathbf{p}_o - \mathbf{p}\|$ being the distance between $\mathbf{p}_0$ and $\mathbf{p}$. ε_0 is the dielectric constant and ε_r is a material specific constant.

In many cases, a vector field varies with time, *i.e.*, the vector quantity does not only vary at different locations but it is also dependent on time.

Definition 2 (vector field, time-dependent)
Given a region R in space and a vector space V, a time-dependent vector field is a function $F : R \times \mathbb{R} \to V$, that maps a given location $\mathbf{x}$ and time t to a vector value $F(\mathbf{x}, t) = \mathbf{v}$.

2.2 Multidimensional search trees

Given a set of positions $P = \{\mathbf{p}_1, \ldots, \mathbf{p}_N\}$, it is often necessary to organize these points in a way that enables efficient queries for positions with a given property. *E.g.*, given the set P of positions and an additional position $\mathbf{p}$, it might be necessary to locate the n points, that are closest to this position. *Multidimensional search trees*, also known as *k-dimensional trees* (k-D-trees) are a generalization of *binary search trees*[1] and were introduced by Bentley [1] as an efficient data structure for queries on data sets containing multiple search keys. This property makes them suitable for geometric applications. *E.g.*, positions contain coordinates for each dimension. These coordinates can be viewed as search keys.

Like a binary search tree, a k-D-tree successively partitions a data set into subsets, which are, in the best case, of approximately the same size. In a k-D-tree the data set is split alternatingly in the different dimensions. This means in the 2D case that the first level of the tree splits according to the x-coordinate, the second level according to the y-coordinate, the third level according to the x-coordinate, etc. This cycle continues, until a node only contains one single point. This is

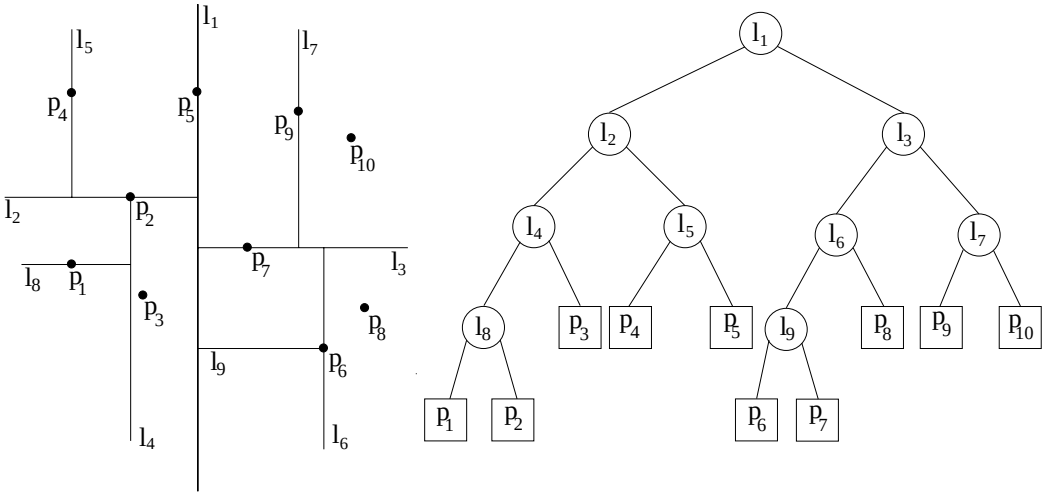


Figure 2.1: Subdivision of the plane by a k-D-tree.

illustrated in Figure 2.1. At the root, the point set P is split into two subsets of roughly the same size. The splitting line (l_1) is stored in the root node along with two pointers to the positions left and right from this line. These subsets are each split along a horizontal line (l_2 and l_3). We note that for each subset a different horizontal line is used. This is done, because the subsets are now considered independent of each other, allowing recursive implementations. Furthermore for each subset a different splitting line yields an even partition in approximately equal-sized subsets. In each step either a horizontal or vertical line is used as splitting line. Because of this, one coordinate (x- or y-coordinate, depending on whether a horizontal or vertical line is used) is sufficient to describe this line. Choosing the median coordinate of the point set, *i.e.*, the

[1] Refer to any textbook on data structures, *e.g.*, [39] or [19]

coordinate for which there is an equal number of points having a smaller value and points having a larger value for this coordinate, is chosen. This yields the desired even partition of the set of points. This concept can easily be generalized to higher dimensions. Instead of alternating

Algorithm 1 Build k-D-tree(P, $depth$)

Input: A set of points P and a current depth $depth$
Output: The root of a k-D-tree storing P

```
if |P| = 1 then
  return a leaf storing this point;
else
  splitDimension ← depth mod k;
  Compute median coordinate medianCoord in dimension splitDimension;
  P_1 ← ∅; P_2 ← ∅;
  for all p ∈ P do
    if Coordinate splitDimension of p < medianCoord then
      P_1 ← P_1 ∪ {p};
    else
      P_2 ← P_2 ∪ {p};
    end if
  end for
  ν_left ← Build k-D-tree(P_1, depth + 1);
  ν_right ← Build k-D-tree(P_2, depth + 1);
  Crate node ν storing splitDim and medianCoord with ν_left and ν_right as left and right
  child;
  return ν;
end if
```

between two dimensions it is possible to cycle through the dimensions. The corresponding algorithm for constructing a k-D-tree is illustrated as Algorithm 1. Further improvements can be achieved when, instead of simply cycling through the dimensions in each step, the optimal dimension for splitting is chosen. This was done by Friedman *et al.* [8].

When the k-D tree is constructed, it can be used to efficiently perform queries for these points. *E.g.*, for a given point the n closest neighbor points can be located. An algorithm for performing this kind of query is discussed in [8].

This Diplomarbeit uses a finished implementation of k-D-trees (refer to Appendix A.3) which is discussed in detail in [27]. It is an improvement of the methods discussed in [8].

2.3 Scattered data

Vector field data can be given in the form of analytical functions or as discretized fields.

Definition 3 (scattered data)
Given a set of positions $\mathbf{x}_i$, $i = 1, \ldots, N$, *called sites, a set of scattered data is a relation that*

maps each site $\mathbf{x}_i$ *to a value* $v_i = F(\mathbf{x}_i)$*. This can be expressed as a set of* N *tuples* $(\mathbf{x}_i, v_i)$ *containing the sites and the corresponding values.*

For scattered data sets of vectors, the v_i are vectors and will be written as $\mathbf{v_i}$. The vector field is given as a set of tuples $(\mathbf{x}_i, \mathbf{v_i})$. In the context of this work, this tuple of position and associated vector is called *sample.*

Definition 4 (sample)
A single tuple $(\mathbf{x}_i, v_i)$ *of a set of scattered data is called sample.*

Scattered data interpolation deals with the problem of estimating values for the vector field at arbitrary locations.

A "classical" method to interpolate the data between sites is *Shepard's method* (see [7]). It is an inverse distance weighted method, meaning that the influence a of sample i on the value of the interpolant in a given position $\mathbf{p}$ is inversely proportional to the distance between the position $\mathbf{p}$ and the site $\mathbf{x}_i$ of the sample. This interpolant is computed as follows:

$$F(\mathbf{p}) = \frac{\sum_{i=1}^{N} \frac{v_i}{\|\mathbf{p}-\mathbf{x}_i\|^2}}{\sum_{i=1}^{N} \frac{1}{\|\mathbf{p}-\mathbf{x}_i\|^2}} \quad . \tag{2.2}$$

The evaluation of this interpolant does not involve the solution of an equation system. However, in most cases the quality of the achieved results is unsatisfactory.

Better results can be attained by using *Hardy's multiquadric* method (see [12, 13]). This interpolant is computed by evaluating

$$F(\mathbf{p}) = \sum_{i=1}^{N} \alpha_i \sqrt{\|\mathbf{p} - \mathbf{x}_i\|^2 + R^2} \quad , \tag{2.3}$$

where $R^2 > 0$ is a user-specified positive constant. The α_i can be computed by solving the system of linear equations resulting from equating $F(\mathbf{p}_i) = v_i$. For scalar values, this results in a system of $N \times N$ equations. For vector fields with vectors in $\mathbb{R}^m$, the α_i are vectors in $\mathbb{R}^m$ as well and the resulting system will consist of $Nm \times Nm$ linear equations. The results achieved with Hardy interpolation are generally very good.

The method can be further improved by using a localized Hardy interpolation. This modifies the general Hardy approach by using only a subset of the samples for computing the interpolant at any given position. For any given position $\mathbf{p}$ only the k nearest samples (using the Euclidean distance of site positions) are used for computing the interpolant. Given a position $\mathbf{p}$ and a set of scattered data $(\mathbf{x}_i, v_i)$ the resulting algorithm for a localized Hardy interpolation works as illustrated in Algorithm 2.

One problem using these scattered data methods is depicted in Figure 2.2. Since only samples and no connectivity information is given, it is not possible to derive the position of the wing from the scattered data. Concerning the interpolation, the geometry of the wing is completely disregarded. If, for example, a position within the wing is interpolated, the result is not that there is no flow in that region, but rather an interpolated value taking samples of opposite sides

Algorithm 2 Localized Hardy interpolation

Input: A set of samples S and a position $\mathbf{p}$
Output: The vector value $\mathbf{v}$ of the vector field in $\mathbf{p}$
Use a k-D-Tree to find the k nearest sites to $\mathbf{p}$ in the data set;
Use the corresponding samples to solve the $km \times km$ system of linear equations for the α_i resulting from equation 2.3;
Use equation 2.3 to compute the vector value at this position;
return this value;

of the wing into account yielding an incorrect and meaningless value for this location. Even worse is the fact that for interpolating values on one side of the side of the air wing, samples on the opposite side can be used, distorting the result and causing incorrect values.

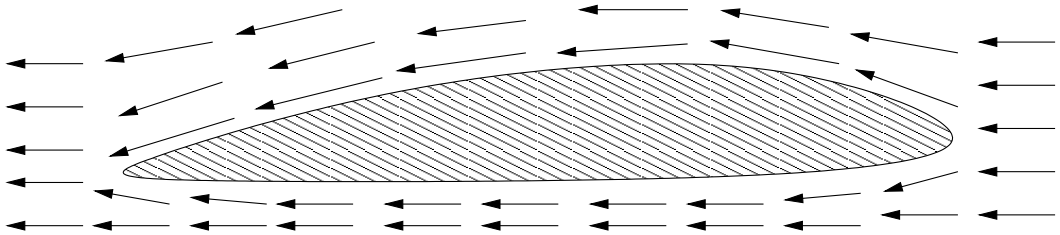


Figure 2.2: Discrete flow field around an airplane wing.

There are extensions to scattered data interpolation that place *barrier points* along the wing. These are points, that prevent the samples from one side to have influence on interpolation results on the other side. However, this requires additional information that often is not given.

2.4 Triangular and tetrahedral meshes

Another method to avoid the problem depicted in the previous paragraph is using grid-based methods for interpolation. Grids define the connectivity between points. This is done by considering the points as vertices of polyhedral lattices. This yields a tessellation of the region, in which the vector field (or any other field) is defined, into sub-regions, called *cells* or *elements*. Essentially, it is possible to distinguish between two fundamental grid types:

- *Structured grids* that are usually generated by a combination of *transfinite interpolation* (TFI) and solving elliptic or hyperbolic systems of partial differential equations (PDE). These grids are generally associated with grids of cells that are topologically equivalent to a square (2D space) or cube (3D space).
- *Unstructured Grids* are typically generated using a triangulation/tessellation algorithm (*e.g.*, a *Voronoi diagram* or a *Delaunay triangulation* approach).

There are variations of these fundamental grid types including hybrid approaches. A more thorough discussion of this topic, that constitutes an own area of research can be found in Chapter 3 of [30].

The advantage of these grids is that in many cases they are supplied with the data, since, *e.g.*, in CFD these grids are necessary to generate the data set. If the data set originates from measured data or the grid is not available, it is necessary to generate one. In that case the same problems in determining the geometry arise, as with the barrier points in scattered data interpolation.

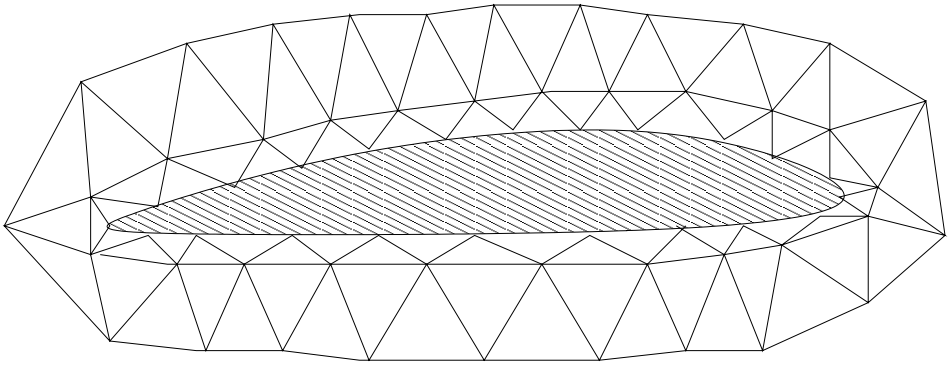


Figure 2.3: Triangular grid around an airplane wing.

If a grid is given, the value at a point can be estimated by locating the cell containing the point for which to interpolate. If linear interpolation is used (this is the case for most of the visualization techniques currently used) the value can be directly computed from the values at the vertices of the cell. If some higher-order interpolation scheme is used, the values of the adjacent cells have to be considered as well. Since the grid specifies which cells and vertices are adjacent, the danger of using samples, *e.g.*, from the opposite side of the airplane wing is avoided. This is shown in Figure 2.3. This Figure shows the airplane wing from the previous section with an unstructured grid consisting of triangles. As the Figure shows, only the region around the wing is filled with triangles. It is easy to determine, whether a given point is in the region in which the vector field is defined. Furthermore, the triangles connect only points that are adjacent. So there are no connections within the wing. This prevents samples from the opposite side of the wing being used, when the interpolated value for a position within the vector field is computed.

This work deals only with unstructured grids, or, to be more specific, unstructured grids composed of tetrahedral cells. Furthermore, the used data sets only consisted of data points. So the generation of the grid necessary for the interpolation is discussed as well.

2.5 Voronoi diagrams, Delaunay triangulations, and Gabriel graphs

The *Voronoi diagram*, the *Delaunay triangulation*, and the *Gabriel graph* have classically been studied in the area of *computational geometry*. More thorough and formal descriptions of these terms can be found in [2] and [35]. An in-depth discussion of Voronoi diagrams and Delaunay triangulations can be found in [33].

Definition 5 (convex)
A subset (or region of space) $R \subset \mathbb{R}^n$ is called convex, if and only if for any pair of points $\mathbf{p}, \mathbf{q} \in R$ the line segment $\overline{\mathbf{pq}}$ connecting $\mathbf{p}$ and $\mathbf{q}$ is completely contained in that region.

Definition 6 (convex hull)
Given a set of points $\mathbf{p}_i$, $i = 1, \ldots, N$, the convex hull of this set is the smallest convex region in space, containing all the points.

Definition 7 (convex combination)
Given a set of points $\mathbf{p}_i$, $i = 1, \ldots, N$, a convex combination of this set of points is a linear combination of the points of the form

$$\sum_{i=1}^{N} \lambda_i \mathbf{p}_i, \quad \text{where } 0 \leq \lambda_i \leq 1 \text{ and } \quad \sum_{i=1}^{N} \lambda_i = 1 \quad . \tag{2.4}$$

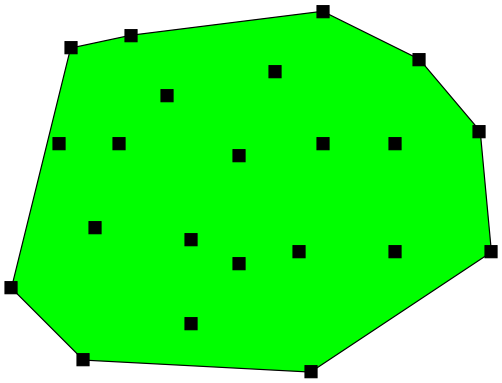


Figure 2.4: Convex hull of a set of points (2D case).

The convex hull is equal to the set of all possible convex combinations. An example of a convex hull is shown in Figure 2.4.

A Voronoi diagram (see Figure 2.5) defines a partition of space. Given a set of points $\mathbf{s}_i$ (again called *sites*) with $i = 1, \ldots, N$, a Voronoi diagram partitions space into regions called *Voronoi cells* C_j, $j = 1, \ldots, N$, around the sites. Each Voronoi cell contains one site and all points of space that are nearest to this site (*i.e.*, the Euclidean distance to this site is smaller than the Euclidean distance to any other site).

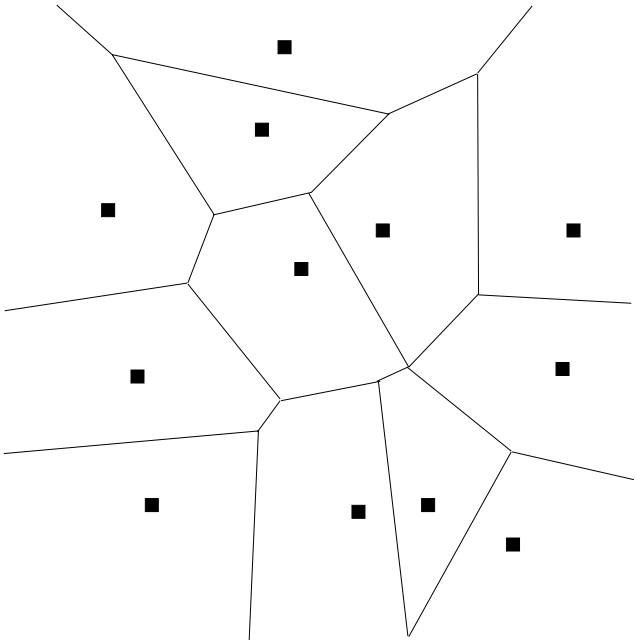


Figure 2.5: Voronoi diagram for a given set of points in the plane.

Definition 8 (Voronoi diagram)
Given a set of sites $\mathbf{s}_i$*,* $i = 1, \ldots, N$*, a Voronoi diagram is the partition of* $\mathbb{R}^n$ *into disjoint convex Voronoi cells* C_i *satisfying the condition*

$$\mathbf{p} \in C_i \Rightarrow \forall_{j=1,\ldots,N;\, j \neq i} : \|\mathbf{p} - \mathbf{s}_i\| < \|\mathbf{p} - \mathbf{s_j}\| \tag{2.5}$$

The Voronoi diagram for any given set of sites is unique.

During the previous section grids were discussed. One method to generate a grid for a given set of points $\mathbf{p}_i$ is to compute a *triangulation* of the points. This yields an unstructured grid, that can be used for the CFD simulation or to interpolate values between the sites.

Definition 9 (triangulation)
Given a set of points $\mathbf{p}_i \in \mathbb{R}^n$*, a triangulation is a decomposition of the convex hull of these points into* n*-dimensional simplices (triangles in two dimensions and tetrahedra in three dimensions) whose defining points (vertices) coincide with the* $\mathbf{p}_i$*. The simplices may not intersect one another and must only share boundary elements such as vertices, edges, or faces.*

If two simplices share a boundary element, they share it completely and not only parts of it (*e.g.*, two tetrahedra share either a complete face or no face at all. They do not share parts of faces).

It is easy to see that for a given set of points no unique triangulation exists. In order to choose a triangulation from all possible triangulations it is necessary to find a way to distinguish between quality of different possible triangulations. Figure 2.6 shows a set of points in $\mathbb{R}^2$ and two different triangulations for this set of points. The first triangulation partitions the convex hull into triangles of about the same size, whereas the second triangulation contains several

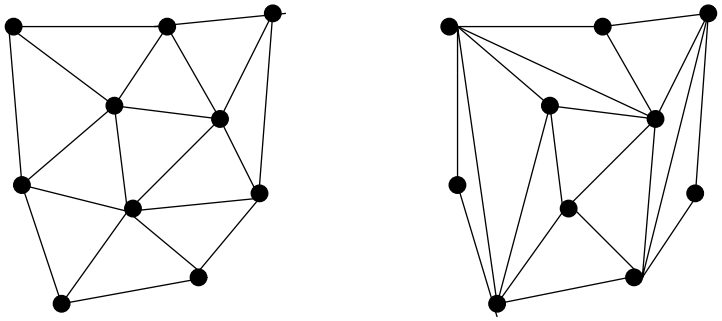


Figure 2.6: Two triangulations for the same set of points.

large but narrow triangles. For interpolation and computational purposes the used triangulation should be as "uniform" as possible concerning triangle shape. In order to develop an algorithm to generate an "uniform" triangulation, it is necessary to find criteria that ensure this desired property. One possibility to ensure this is the *minimum interior angle* criterion.

Definition 10 (minimum interior angle, triangle)
Let $\alpha_j^{(T)}$, $j = 1, 2, 3$, be the interior angles of a triangle T. Then $\alpha_{min}^{(T)} = \min(\alpha_j^{(T)} | j = 1, 2, 3)$ denotes the minimum interior angle of this triangle.

Definition 11 (minimum interior angle, triangulation)
Let $\mathcal{T}$ be a triangulation of the sites $\mathbf{x}_i \in \mathbb{R}^2$, $i = 1, \ldots, N$, consisting of the triangles $T_1, \ldots, T_K$. The minimum interior angle of this triangulation is $\alpha_{min}^{\mathcal{T}} = \min(\alpha_{min}^{(T_l)} | l = 1, \ldots, K)$

Definition 12 (minimum interior angle criterion)
Let Γ be the set of all possible triangulations for the sites $\mathbf{x}_i \in \mathbb{R}^2$, $i = 1, \ldots, N$. A triangulation $\mathcal{T} \in \Gamma$ is optimal in regard to the minimum interior angle criterion, if and only if it has the largest possible minimum interior angle, i.e.,

$$\forall \mathcal{T}' \in \Gamma : \alpha_{min}^{\mathcal{T}'} \leq \alpha_{min}^{\mathcal{T}} \quad . \tag{2.6}$$

In 2D space, a triangulation satisfying the minimum interior angle criterion is an optimal triangulation. In order to construct a triangulation that achieves this optimum it is useful to consider another criterion.

Definition 13 (circumsphere)
Given a simplex in $\mathbb{R}^n$ the circumsphere is the n-dimensional hypersphere containing the $n + 1$ defining points (vertices) of the simplex.

Definition 14 (Delaunay triangulation)
A triangulation of a set of sites $\mathbf{x}_i \in \mathbb{R}^n$, $i = 1, \ldots, N$, is a Delaunay triangulation if the circumsphere of each simplex contains no other site than the vertices of that simplex.

Definition 15 (co-spherical)
A set of points $\mathbf{x}_i \in \mathbb{R}^n$, $i = 1, \ldots, N$, is co-spherical if there is a n-dimensional hypersphere containing all points of this set.

In 2D space, the Delaunay triangulation satisfies the minimum interior angle criterion and thus is an optimal triangulation for that set of points. In higher dimensions, the Delaunay triangulation is not necessarily optimal. However, it is commonly used for grid generation. Another property of the Delaunay triangulation is that it is unique if the sites satisfy the condition that no subset containing $n + 2$ of the original site is co-spherical. If a set of sites satisfies this criterion the points are in *general position.* This is illustrated in Figure 2.7. The point set shown

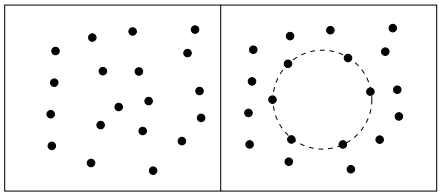


Figure 2.7: Point set in general position and point set not in general position.

on the left hand side is in general position. The point set on the right hand side is not in general position, because six points lie on a circle.

Definition 16 (general position)
A set of positions $\mathbf{x}_i \in \mathbb{R}^n$, $i = 1, \ldots, N$, is in general position if it contains no subset of $n + 2$ or more co-spherical points.

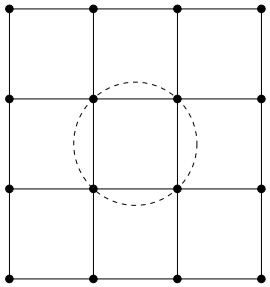


Figure 2.8: Points on a rectilinear grid are not in general position.

Unfortunately, point sets not in general position are rather common. Commonly data sets are sampled on a rectilinear grid. In that case, adjacent points form quadrilaterals (in 2D space) or cubes (in 3D space) and sets of 4 (2D space, refer to Figure 2.8) or 6 points (3D space) lying on a circle or sphere exist. In these cases the Delaunay triangulation is not unique and many algorithms for Delaunay triangulation fail. Since in these cases any triangulation of the points satisfies the Delaunay criterion, an arbitrary triangulation can be chosen. This is illustrated in

Figure 2.9. There are no reasons, why either triangulation should be chosen instead of the other one. One way to accomplish a unique Delaunay triangulation, even when the points are on a rectilinear grid, is to *jitter* the points, *i.e.*, add a small random value to each coordinate. This value has to be small enough to preserve the position of the points in relation to each other, but sufficiently large to resolve the ambiguities in the triangulation.

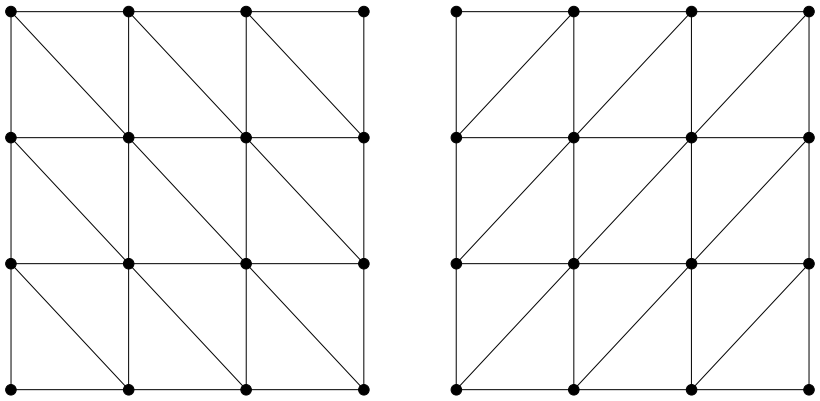


Figure 2.9: Ambiguity when triangulating points on a rectilinear grid.

Delaunay triangulations and Voronoi diagrams are closely connected. This connection is defined via the *Delaunay graph.*

Definition 17 (Delaunay graph)
Given a set of positions $\mathbf{x}_i \in \mathbb{R}^n$, $i = 1, \ldots, N$, and the Voronoi diagram for these sites, the Delaunay graph is the graph that results from connecting all sites whose Voronoi cells are adjacent to each other.

If the positions are in general position, then the edges in the Delaunay graph correspond to the edges in the Delaunay triangulation. If the set contains $n + 2$ or more co-spherical points the Delaunay graph can be transformed into a Delaunay triangulation by adding edges so that the resulting graph consists of edges of simplices.

Given the definition of the Delaunay graph via Voronoi diagrams it is easy to see that it represents the intuitive neighborhood relation. If two points are connected by a Delaunay edge then a human would also consider them to be adjacent or neighbors. Because of this property the Delaunay graph is also said to represent a natural adjacency relation.

The *Gabriel graph* is a subset of the Delaunay graph that is easier to compute.

Definition 18 (Gabriel graph)
The Gabriel graph is the subgraph of the Delaunay graph consisting of all Delaunay edges, that intersect the border between the Voronoi cells containing the points they connect.

It can be shown that the Gabriel graph can be constructed by connecting all points $\mathbf{x}_i$ and $\mathbf{x}_j$ ($i \neq j$) with the property that the hypersphere with diameter $\overline{\mathbf{x}_i\mathbf{x}_j}$ contains no other point in its interior.

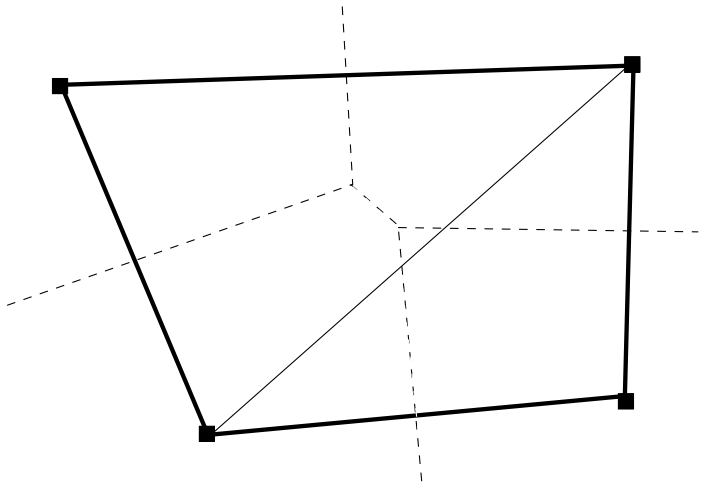


Figure 2.10: Delaunay and Gabriel graphs of four points.

Figure 2.10 shows a set of points in 2D and the difference between the Delaunay and the Gabriel graph. The points are shown as filled square. The corresponding Voronoi diagram is denoted with dashed lines. The Delaunay graph consists of the bold solid lines and the solid line. The Gabriel graph consists only of the bold lines; it is missing one of the edges of the Delaunay graph.

2.6 Triangulation-based interpolation

Given a triangular grid consisting of N samples and a triangulation of these points, it is possible to use this information to interpolate values at any location within the convex hull of the locations. As mentioned before, in visualization the most commonly used type of interpolation is *linear interpolation.* In this case, the value for a given point $\mathbf{p}$ can be computed by using only the samples at the vertices of the simplex containing $\mathbf{p}$. Given this simplex, the interpolation value can be computed using *barycentric coordinates* of $\mathbf{p}$.

Definition 19 (Barycentric coordinates)
Given a simplex in n-dimensional space with the vertices $\mathbf{x}_i$, $i = 1, \ldots, n+1$, each position $\mathbf{p}$ can be represented as a linear combination of the vertices, i.e.,

$$p = \sum_{i=1}^{n+1} \beta_i \mathbf{x}_i \quad . \tag{2.7}$$

This linear combination is called barycentric coordinates. Each position expressed as barycentric coordinates is within the simplex if and only if the β_i satisfy the condition

$$0 \leq \beta_i \leq 1 \quad . \tag{2.8}$$

Barycentric coordinates satisfy the equation

$$\sum_{i=1}^{n} \beta_i = 1 \tag{2.9}$$

In 2D space, the barycentric coordinate β_i can be computed as the ratio of the area of the triangle that results from replacing $\mathbf{x}_i$ with $\mathbf{p}$ to the area of the original triangle.

$$\beta_1 = \frac{A_{\triangle \mathbf{p x_2 x_3}}}{A_{\triangle \mathbf{x_1 x_2 x_3}}}; \qquad \beta_2 = \frac{A_{\triangle \mathbf{x_1 p x_3}}}{A_{\triangle \mathbf{x_1 x_2 x_3}}}; \qquad \beta_3 = \frac{A_{\triangle \mathbf{x_1 x_2 p}}}{A_{\triangle \mathbf{x_1 x_2 x_3}}} \tag{2.10}$$

It is important to use the *directed area.* This is necessary in order to satisfy condition 2.9. In the 3D case, the barycentric coordinates can similarly be computed as ratio of directed volumes of the tetrahedra resulting from replacing a vertex with $\mathbf{p}$ to the directed volume of the original tetrahedron. Computing the barycentric combination from the ration of areas in the 2D case is

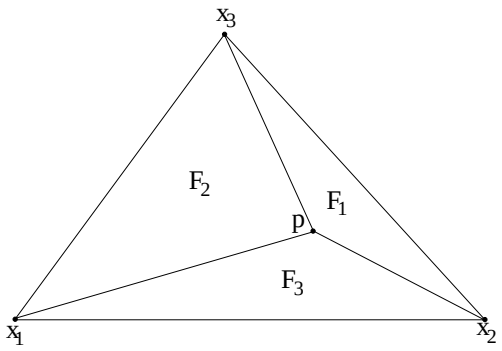


Figure 2.11: Computing the barycentric coordinates as ratios of areas.

shown in Figure 2.11. The areas F_1, F_2, and F_3 of the three triangles resulting from substituting $\mathbf{x}_i$ with $\mathbf{p}$ have to be divided by the area of the complete triangle in order to get the barycentric coordinates β_i.

If the values of the vertices $\mathbf{x}_i$ are $\mathbf{v_i}$, $i = 1, \ldots, n+1$, then the interpolated value at the position $\mathbf{p}$ can be computed as

$$\sum_{i=1}^{n+1} \beta_i \mathbf{v_i} \quad , \tag{2.11}$$

where the β_i are the barycentric coordinates of $\mathbf{p}$.

The remaining problem is to locate the simplex containing the point $\mathbf{p}$. The simplest possibility is to search linearly in all simplices and find the simplex containing $\mathbf{p}$. This can be done by computing the barycentric combination of $\mathbf{p}$ for each tetrahedron and check whether condition 2.8 is satisfied. As soon as this simplex is found, the already computed barycentric combination can be used to compute an estimated value. This can be done with time complexity $O(n_{\text{Triangles}})$.

A more efficient approach is presented by E.P. Mücke *et al.* [28]. They describe an algorithm (simple walk-through) to locate a position in a Delaunay triangulation. First, a start simplex is chosen. Then a line segment L from one of the vertices of this start simplex to the query point $\mathbf{p}$ is computed. The simplex containing $\mathbf{p}$ is located by "walking along" L, traversing all simplices intersected by L. The paper proves that the cost to find the simplex has an expected value that is proportional to $O(n_{\text{Triangles}}^{\frac{1}{d}})$ and presents a further improvement of the algorithm (improved jump-and-walk).

In this work, a variation of the basic walk-through algorithm is used on a Delaunay triangulation to locate the tetrahedron containing a point for which a value must be estimated. First, like in the simple walk-through approach, a tetrahedron of the mesh is selected. A reasonable choice would be the tetrahedron that contains the "last queried point", however the algorithm works even if an arbitrary tetrahedron is chosen. Now, instead of computing a line segment between one of the vertices of the current tetrahedron and the queried position, the barycentric combination for the queried point in regard to the current tetrahedron is computed. If $\mathbf{p}$ is inside the tetrahedron this can be decided by means of the barycentric combination and the interpolation value can be computed.

However, if condition 2.8 is not satisfied and the queried position is not within the current tetrahedron, then the barycentric combination can be used to determine beyond which face of the current tetrahedron the queried point is located.

If condition 2.8 is violated, there exits at least one negative barycentric coordinate ($\beta_i < 0$). This is a result from condition 2.9. If condition 2.8 is violated and there were no negative barycentric coordinate, that would mean that all $\beta_i > 1$, and thus $\sum_{i=1}^{n+1} \beta_i > 1$ which violates condition 2.9. A negative barycentric coordinate β_i indicates that the corresponding vertex $\mathbf{x}_i$ and the queried point $\mathbf{p}$ lie on opposite sides of the triangular face formed by the remaining vertices $\mathbf{x}_j \;\; j \neq i$ of the tetrahedron. Figure 2.12 illustrates this situation. The point $\mathbf{p}$ lies

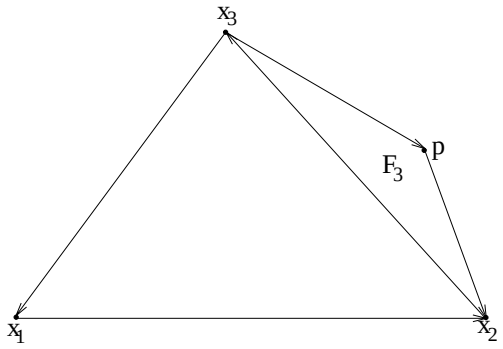


Figure 2.12: Opposite sign of the directed area for a point outside the triangle.

beyond the edge corresponding to the barycentric coordinate β_1. This entails that the triangle resulting from replacing $\mathbf{x_1}$ by $\mathbf{p}$ has the opposite orientation as the original triangle (in the Figure the orientation is given by the arrows at the end of the triangle edges) and thus the opposite sign for the directed area.

If the queried point $\mathbf{p}$ is not within the current tetrahedron, then the modified algorithm uses this fact and chooses the first negative barycentric coordinate. It then chooses the tetrahedron beyond the face opposite to the corresponding tetrahedron vertex $\mathbf{x}_i$ as next step in the "walk" toward the query point. Algorithm 3 illustrates this for the case of a tetrahedral mesh.

Algorithm 3 Modified simple walk-through for Delaunay triangulations in 3D space

Input: A triangulation T and a query point $\mathbf{p}$
Output: An interpolated value $\mathbf{v}$

```
Let currTet point to the tetrahedron that contained the last queried point or an arbitrary
tetrahedron;
Compute the barycentric combination of p in currTet;
while ∃_{i∈{1,2,3,4}} β_i < 0 ∨ β_i > 1 do
    Find the tetrahedron nextTet beyond the face opposite to vertex p_i;
    if nextTet is undefined then
        return Undefined;
    end if
    currTet ← nextTet;
    Recompute the barycentric combination of p in currTet;
end while
return the value computed according to equation 2.11;
```

In the implementation used for this work, the class "`TetVertex`", which is used to represent the tuple of position and vector information for all sample points also includes a list of tetrahedra that contain that vertex. This makes it possible to quickly access the tetrahedra adjacent to the current tetrahedron. Finding the tetrahedron that lies beyond a face defined by three vertices of the current tetrahedron is done by computing the intersection[2] of the three lists of tetrahedra containing the three vertices defining the face.

It would be possible to further differentiate between cases and only consider tetrahedra that share only an edge (if there are two negative barycentric coordinates) or a vertex (if there are three negative barycentric coordinates). However, it is doubtful that the reduced length of the "walk" would outweigh the additionally imposed computational cost. The advantage of the current implementation is that a minimum of information is calculated which is not used for the actual interpolation. Apart from computing the barycentric coordinates that have to be computed in any case, it is only necessary to maintain the list of vertex owners (tetrahedra containing the given vertex) and compute the tetrahedron beyond a given face.

If the "queried point" is outside the tetrahedral mesh, the "walk" ends in a tetrahedron, that does not contain the point and has no neighboring tetrahedra beyond the faces in direction of the "queried point". In that case the value for the "queried point" is undefined.

An example for this algorithm (in 2D space to improve clarity) is shown in Figure 2.13. It shows a walk toward the query point in a Delaunay triangulation. In each step, the current triangle is drawn in green. The query point is marked with "x", and the edges for which the barycentric combination criterion is satisfied are marked red.

[2] In the actual implementation, a hash-table is used to speed up the implementation.

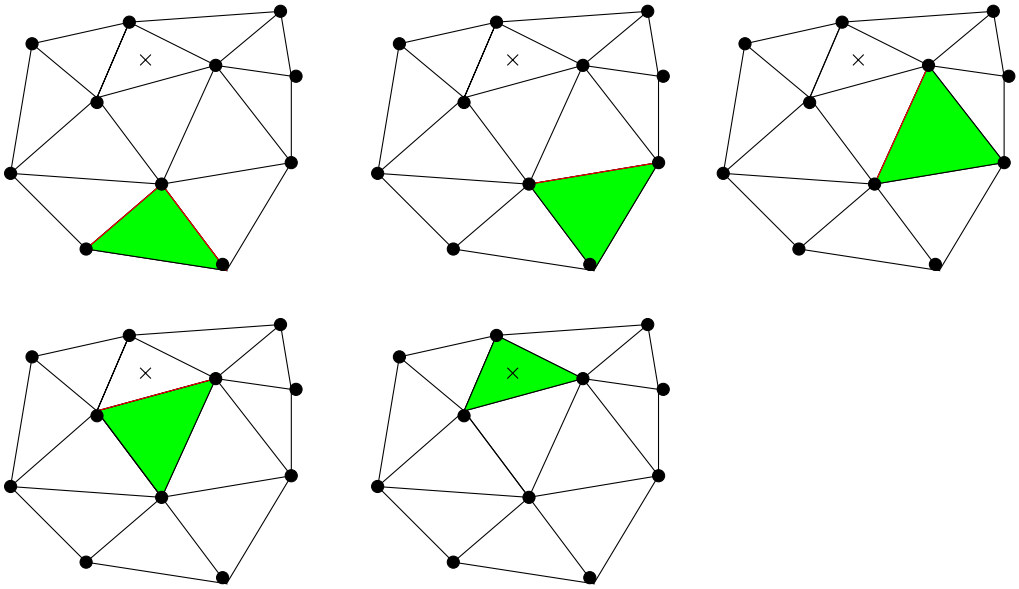


Figure 2.13: “Walking” toward the query point.

2.7 Clustering

Clustering is a classical data analysis technique, frequently used in statistics, that has been thoroughly studied. It has been used in a wide variety of fields, including data mining. The main idea is to generate clusters of objects of similar or equal properties enabling one to recognize underlying structures. These structures can be used to generate an approximation of the data keeping its main features. A more comprehensive description of clustering can be found in [11].

2.8 Clustering for surface reconstruction

Clustering was successfully used as an innovative technique for surface reconstruction by Heckel *et al.*, see [14, 15], to reconstruct a surface from a set of points. Initially, all points are assigned to one cluster, and an error for this cluster is computed. In each following step, the cluster with the largest error is chosen and split. In surface reconstruction a sensible choice for this error is the deviation of the cluster points from a flat surface. The cluster selected using this measure is split in the direction of maximum extension. All points in adjacent regions to the newly split clusters are *reclassified* (*i.e.*, their association to a cluster is re-evaluated) and assigned to the nearest cluster with respect to Euclidean distance from the given *cluster centers*, also called *cluster representants*. New errors for the affected clusters are computed, and the splitting process is repeated until a given number of clusters has been created or until all the error of all clusters is below a given limit.

In an efficient implementation of this approach, only points in clusters adjacent to the newly created clusters should be considered during the reclassification step. This imposes the need

to determine a neighborhood relationship between clusters to find all neighbor clusters of the new cluster and subject their sample points to reclassification. The original implementation of the clustering algorithm used the Gabriel graph to determine this relationship. Recent research showed that using this graph does not yield completely satisfactory results, and modifications were made taking additional clusters, besides those connected via an Gabriel edge, into account. However, using the Delaunay instead of the Gabriel graph promises to yield even better results.

2.9 Using clustering to simplify vector fields

The main issue in utilizing a clustering algorithm for vector field simplification are choosing the distance function and defining new cluster centers (or cluster representants) for the split process. In trying to find an answer to the question what choice of distance functions yields the best results and whether to use a single distance function for everything, or a combination of different distance functions, three approaches have been studied.

To understand the different approaches it is necessary to examine first for what purposes the distance function is used during clustering. These purposes are:

1. to determine which cluster should be split;
2. to determine the locations of the new cluster centers after the split process; and
3. to associate the samples with clusters, *i.e.*, to find out which is the closest cluster for a specific sample.

Given these objectives for a distance function, the following approaches have been examined:

1. Use the same distance function for all purposes. This distance function must depend on both position part and vector part.
2. Distinguish between position and vector part and use two different distance functions, one only dependent on the position part and one only dependent on the vector part of a sample.
3. Use a **two-pass** approach. In one pass, use a distance function depending only on the position part, and in the other pass a distance function depending only on the vector part.

2.10 The definition of distance for vector fields

In order to apply clustering to vector fields and associate the vectors of a vector field with different clusters it is necessary to define a distance function for samples in this vector field. Looking at the vector field as a set of scattered data, each sample of the data set consists of two distinct parts,

1. the position of the sample and

2. the vector value at this sample's position.

In this and the next chapter, it is assumed that the vector value is a vector in Euclidean space (*i.e.*, being described by a length and direction) and not just a collection of scalar values. If vector fields of this kind can be approximated it should be rather straightforward to further generalize this approach to arbitrary vectors by choosing a distance function which might be application-specific. Furthermore, all examples in this chapter are vector fields in 2D space. This is due to the fact that it is easier to study the effects of clustering on these vector fields. The underlying mathematical and algorithmic principles work for vector fields in 3D space as well. The obvious choice for the distance of the position part of two samples is the Euclidean distance between the two positions of the samples.

To determine the distance between the vector part of two samples the simple Euclidean distance could be used. However, it is desirable to have more control over the distance function. A possible choice is a distance function taking two quantities into account, namely

1. the difference in magnitude between the vectors $\Delta_L = |\|\mathbf{v_1}\| - \|\mathbf{v_2}\||$, and
2. the minimal angle between the vectors, $\angle(\mathbf{v_1}, \mathbf{v_2})$.

The second quantity can be computed using the scalar product,

$$\angle(\mathbf{v_1}, \mathbf{v_2}) = \arccos\left(\frac{\mathbf{v_1} \cdot \mathbf{v_2}}{\|\mathbf{v_1}\|\|\mathbf{v_1}\|}\right) \quad .$$

The null vector $\mathbf{0}$ has no direction. If any one of the two vectors is the null vector, the angle is assumed to be zero and only the length difference is taken into account.

These properties can be normalized to lie between 0 and 1 in order to be comparable. For the length difference l, a function like $1 - \mathrm{e}^{-l}$ can be used, or it can be divided by the maximum of the two vector lengths. The minimum angle can simply be divided by $180°$.

Given the fact that a distance function on a vector field is a combination of different quantities, an important question arises: Is a single distance function sufficient or is it necessary to define a set of distance functions and use these in conjunction to yield the desired results? This will be examined in detail in course of the next chapter.

2.11 Vector field visualization

The aim of finding a compact representation of a vector field is to be able to use it for standard visualization techniques. In this section, basic vector field visualization techniques are reviewed. A more detailed description of these techniques can be found in [37] and [9].

Before the advent of computers and scientific visualization, experiments in wind tunnels were performed to examine the properties of vector field. A simple experiment was to inject particles at certain positions in the flow and trace their path during a given period of time.

If one was interested in the behavior of the particle during a given period of time, one possibility was to inject a particle and take a photograph with an exposure time equal to that period of time of interest. The result were pictures of *path lines*, *streak lines*, and *time lines* that showed paths of particles. Mathematically these are defined as follows:

Definition 20 (path line)
Given a region of space $R \subset \mathbb{R}^n$ and a vector field $F \times \mathbb{R} : R \to \mathbb{R}^m$ with $m = n$, a path line traces a the path of a particle that was injected at time $t_0 = 0$ in the flow at the position $\mathbf{x}(0) = \mathbf{x_0}$. This curve is described by the first-order differential equation

$$\frac{d\mathbf{x}}{dt} = F(\mathbf{x}, t) \quad . \tag{2.12}$$

If the vector field is varying over time, it is also of interest how it behaves at any given moment in time. This is impossible to show in a wind tunnel, because time elapses while a particle is traced and the vector field changes. It is impossible to freeze the vector field in an experiment. On a computer this is no problem at all. In fact, removing the time dependency eliminates the need to access data from different time steps and simplifies the visualization. On a computer it is possible to trace the path that a particle would follow, if the vector field remained constant over time and thus visualize the behavior of the vector field at any given moment. These particle paths are called *stream lines* or "instantaneous path lines".

Definition 21 (stream line)
Given a region of space $R \subset \mathbb{R}^n$, a vector field $F : R \times \mathbb{R} \to \mathbb{R}^m$, with $m = n$ and a time t_0, a stream line traces the path a particle would follow, if the vector field remained constant, i.e., $\forall_{\mathbf{x} \in R\ t \in \mathbb{R}} F(\mathbf{x}, t) = F(\mathbf{x}, t_0)$. This curve is the solution of the first-order differential equation

$$\frac{d\mathbf{x}}{dt} = F(\mathbf{x}, t_0) \quad . \tag{2.13}$$

If the vector field is independent of time, path lines and stream lines are identical. Figure 2.14 shows several stream lines around a plate that was put into a flow. Despite the fact that stream and path lines are one-dimensional objects (*i.e.*, extending only in one dimension and thus having no diameter), they are rendered as cylinders with a given thickness. This simplifies evaluating a light model and allows the use of hardware-accelerated shading. Shading provides the viewer with additional cues about the position of the stream lines in respect to each other and enhances the quality of the visualization considerably.

A possibility to enhance the concept of stream lines is to derive a scalar value from the vector field (*e.g.*, velocity), define a mapping between this scalar value and color space and color the stream lines accordingly. This allows the user to gain further insight in the properties of the vector field.

Stream lines, even if colored according to an additional scalar value, convey only local information about the vector field. They can show the rotation of the flow and its derivatives. They do not visualize global information and structure of the field, like *vorticity* or *flow divergence*.

Stream ribbons are a natural extension of stream lines by "widening" them to ribbons. A stream ribbon is a surface that is supposed to be tangential to the flow and that because of being a two dimensional object reveals additional information about the flow. Stream ribbons can be rendered by computing two adjacent stream lines and connecting them with a mesh. This works as long as the adjacent stream lines remain relatively close to each other. If they diverge the resulting surface loses the property of being tangential to the flow. This happens because

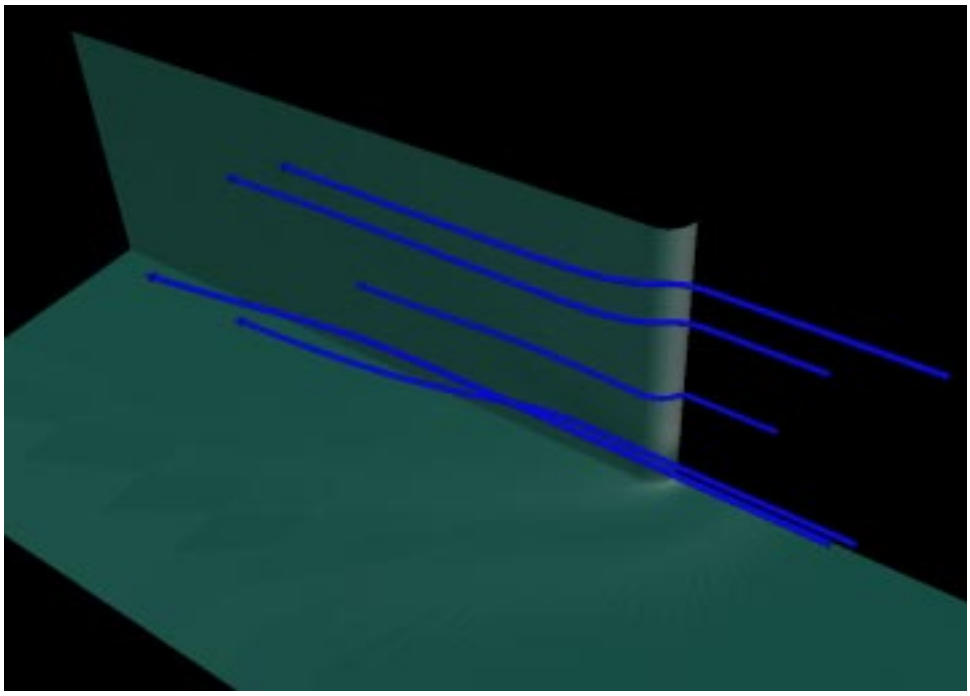


Figure 2.14: Stream lines around a plate. Data set courtesy of NASA Ames Research Center (see Appendix B.3).

the ruled surface connecting two widely separated stream lines is only an approximation to the desired concept.

Stream ribbons can be used to visualize several important flow properties. The vorticity $\boldsymbol{\omega}(\mathbf{x}, t)$ is a measure of the rotation of the vector field expressed as a vector with direction of the axis of rotation and a magnitude denoting the amount of rotation. *Stream-wise vorticity* is the projection of vorticity along the velocity vector at that position, *i.e.*,

$$\Omega(x, t) = \frac{\mathbf{F}(\mathbf{x}, t) \cdot \boldsymbol{\omega}(\mathbf{x}, t)}{\|\mathbf{F}(\mathbf{x}, t)\| \|\boldsymbol{\omega}(\mathbf{x}, t)\|} \quad . \tag{2.14}$$

The amount of twisting of the stream ribbon approximates the stream-wise vorticity of the vector field. The changing width of the stream ribbon shows the *flow divergence*, another property of the vector field. A further generalization of stream ribbons yields the concept of stream surfaces.

Definition 22 (stream surface)
A stream surface is a collection of an infinite number of stream lines passing through a base curve. This base curve specifies the starting points for the stream lines.

Stream surfaces can be approximated choosing a certain finite number of seed points along the base curve (or *rake*), computing the corresponding stream lines and connecting them with a mesh. The points along the rake rake can be specified by the user. Alternatively, the user can

specify a curve and a number and the seed points can be generated automatically. As with stream ribbons, approximating a stream surface with a finite amount of stream lines fails in regions of the vector field, where the divergence is high and stream lines separate. In these regions it is even possible that the surface resulting from the approximation will be self-intersecting, which is physically impossible for stream surfaces. Extensions to the approximation algorithm compute the stream lines concurrently to generating the mesh. If the flow diverges, then additional stream lines are inserted. It is even possible to separate the ribbons of which the surface is composed in regions of extremely high divergence, creating a torn stream surface.

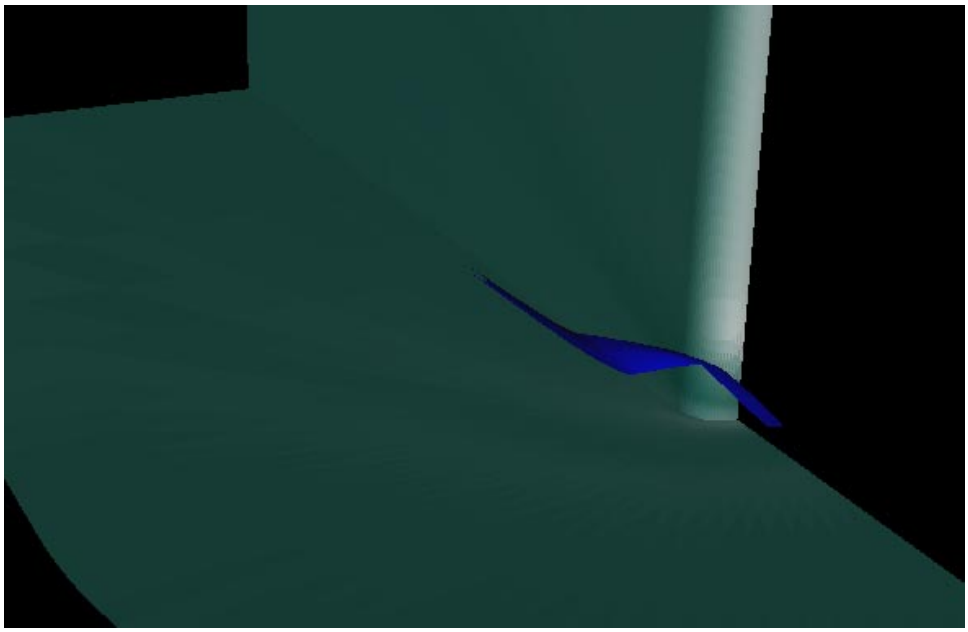


Figure 2.15: Stream surface around a plate. Data set courtesy of NASA Ames Research Center (see Appendix B.3).

Figure 2.15 shows the surface resulting from tracing 20 stream lines along a line between two points and connecting them with a mesh of quadrilaterals. It is a good illustration of the vorticity of the flow at the location of the ribbon.

2.12 Numerical methods

Stream lines are described by first-order differential equations. The solution of these equations is a function describing a parameterized curve. In most cases, it is not possible to compute the analytical solution of the function. Rather a finite number of points along the curve are calculated and connected by line segments. As shown in the last section, a stream line is described by a first-order differential equation and a start value. This problem is referred to as *first-order initial value problem.*

Definition 23 (first-order initial value problem)
The solution to the first-order initial value problem is a function $f(t)$ that satisfies the given conditions:

- *For the argument t_0 the function assumes a given value (initial value)*

$$f(t_0) = x_0 \quad , \tag{2.15}$$

- *and the first derivative of the function is*

$$\frac{df}{dt} = \dot{f}(t) \quad . \tag{2.16}$$

There are several numerical methods to approximate the solution of differential equations of first order. Given the initial value and the first derivative, they compute tuples of function arguments t_i and function values $f(t_i)$. The simplest is *Euler's method*. It starts with the tuple (t_0, x_0). After that, each tuple (x_{i+i}, y_{i+i}) is computed from (t_i, x_i) using the following equations:

$$\begin{aligned} t_{i+1} &= t_i + h \quad \text{and} \\ x_{i+1} &= x_i + h\dot{f}(t_i) \quad , \end{aligned}$$

where h is the step size along the function. This parameter has to be chosen carefully. Choosing a small value for h increases the computational cost, while choosing a large value decreases the accuracy of the result. In general, Euler's method yields poor results.

A commonly used method that yields better results is the *second-order Runge-Kutta method*. It also starts with the tuple (x_0, y_0). Subsequent tuples are computed according to the rules

$$\begin{aligned} t_{i+1} &= t_i + h \quad , \\ k_1 &= h\dot{f}(t_i) \quad , \\ k_2 &= h\dot{f}(t_i + \frac{k_1}{2}) \quad , \text{ and} \\ x_{i+1} &= x_i + k_2 \quad . \end{aligned}$$

Again, the step size h must be chosen carefully. After computing the x_i-values, they can be connected by line segments to form stream lines. As mentioned in the last section, it is better to render cylindrical segments instead of line segments in order to be able to facilitate the use of the rendering hardware to generate fully shaded stream lines. A more detailed description of these methods can be found in [17], [9], and [36].

2.13 Fundamentals of virtual reality

Virtual reality (VR) is a human-machine interaction paradigm. It is based on presenting a *virtual environment* (VE) to a human. VR is the next natural step after the the transition from purely

text-based to *graphical user interfaces* (GUIs), which started in the mid 80s. Current GUIs present the user with a 2D screen and a mouse and are based on a desktop metaphor, using concepts known from an office environment (desktop, files, folders, etc.), creating a more or less intuitive user interface.

Shutter glasses that alternately cover the eyes of a user allow to present different images to both eyes of the user. This makes it possible to present images that extend beyond the two dimensions of the screen. This can be used to provide better cues for the relative position of objects.

Figure 2.16: Head-mounted display.

Head-mounted displays (see Figure 2.16) take this concept one step further by providing two screens, one for each eye. They allow the user to immerse herself/himself completely into a VE. They also alleviate one of the problems that shutter glasses introduce. Since shutter glasses use one display to present images to both eyes of an observer, they have to attain higher frame refresh rates in order to create flickerless images.

Figure 2.17: Boom-based head tracking.

In order to completely immerse a viewer into a VE it is necessary to track the position of her/his head and update the generated images accordingly. If the user turns her/his head in the VE, she/he naturally expects that she/he will be offered a different view of the environment.

This can be achieved by adding a device to the display that tracks the movements of the user's head. This can be done using electronic devices that track the position of a small sensor attached to the head-mounted display, or by mechanical means, like the boom device shown in Figure 2.17.

Immersive displays alone are not sufficient to create the illusion of being immersed in a VE. If the powerful concept of VR is to be used at its full strength, appropriate input devices are necessary as well.

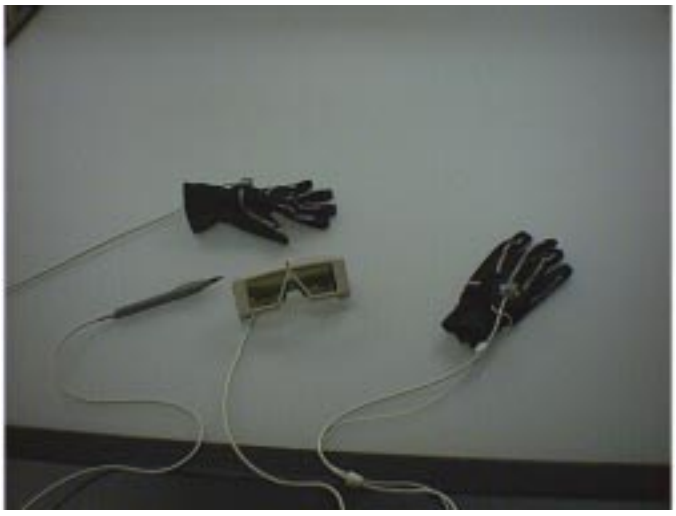


Figure 2.18: VR input devices.

Figure 2.18 shows two examples of VR input devices. The *stylus* is the object shaped like a pen in the lower left corner. It contains an electro-magnetic tracking sensor to query its position and has a button allowing the user to select objects. The *pinch gloves* (upper left and rightmost object) also have sensors tracking their position (the grey rectangular object attached to it) and can detect simple gestures. If the user performs a *pinch* gesture (*i.e.*, she/he brings two of her/his fingers in physical contact), then this is indicated to the application and can be used to trigger actions. More elaborate gloves, allowing to determine the exact location of each finger of the hand, are available. Force-feedback gloves even allow to exert a force on the hand of the user, providing her/him with a feedback of her/his actions and allows her/him to "feel" virtual objects.

The *cave* (shown in Figure 2.19) represents a device that allows a group of people to immerse itself into a VE. It basically consists of a set of translucent projection screens as walls. For each wall, a projector displays a computer-generated image onto that wall. The users wear shutter glasses so that objects can appear to be in front or behind the actual projection screen. VR input devices like those described in the previous paragraph can be used to manipulate the displayed environment.

VR and VE add a whole set of new possibilities to the field of scientific visualization. Instead of generating 2D representations of the visualization, it is now possible to create 3D

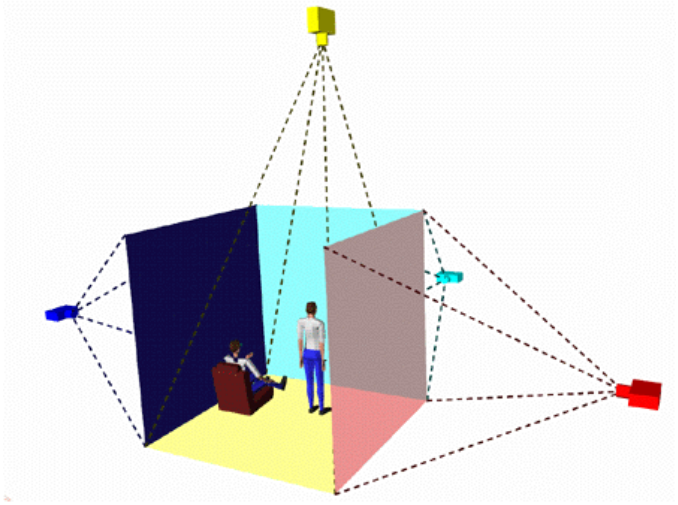


Figure 2.19: The cave.

objects that can be directly manipulated by the user. The possibility to examine the visualized object from different locations by simply walking around allows the user to gain additional insight into the visualized process.

A VE was used in [3] to simulate a wind tunnel. Using the boom-tracked head-mounted display, a user is immersed in an environment simulating a wind tunnel. With this setup, the user can walk through the flow without disturbing it. She/he can place different visualization objects, like stream lines, stream surfaces etc., into the flow and analyze its behavior.

2.14 The Responsive Workbench

Immersive VR allows the user to closely interact with a VE, which is used to communicate with the application. By principle, the user is completely immersed in the environment and cannot perceive any parts of the real world. The disadvantage of this concept is that it hinders collaboration of several users. If a group of engineers wants to discuss the properties of the flow around a certain design, then it is necessary for them to see each other in order to be able to interact with one another. If they are completely immersed in an VE, then it would be necessary to create representations of the engineers in this environment as well. Approaches to this end have already been taken, but lack the full set of possibilities of expression. With current techniques it is just impossible to create a realistic representation of a human in real time.

The *Responsive Workbench*, developed by GMD and Stanford University, takes a different approach to this problem. It creates a *non-immersive environment*[3]. The basic setup is shown in Figure 2.20. It consists of a table with a translucent projection screen. A projector generates an image that is directed to the screen via a mirror. Shutter glasses worn by the users enable

[3] Although a variant is sold under the name *Immersive WorkBench* by Fakespace Inc., Mountain View, CA, this term does not accurately describe the concept.

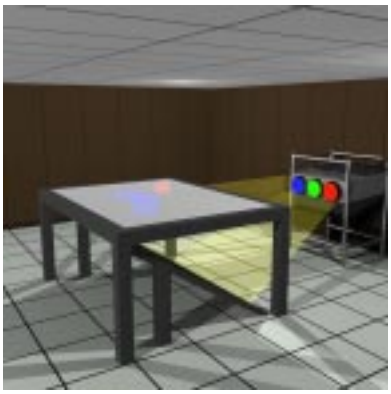


Figure 2.20: The Responsive Workbench.

them to see objects beyond the screen. Users interact with this scenario via VR input devices like those shown in Figure 2.18. The Responsive Workbench is built for VEs that augment the the real environment, which, in this case, comprises of the table. As the name suggests, applications should extend the workbench concept, providing the user with tools she/he can use to alter the scene on the table. In conjunction with the ability to see the other participants, this enables discussions of the VE on the table and facilitates collaboration in this environment. For this work the Immersive WorkBench, a commercial version of the Responsive Workbench, was used. It is an improved version of the original design and has the advantage that the slope of the tabletop, *i.e.*, the projection area, is adjustable. It is similar to a sketch board used by engineers. This can be used to increase the viewing volume by increasing the slope of the projection area. If a horizontal table top is desired, *e.g.*, if physical objects are to be placed on the workbench, then this can be done as well.

2.15 Moving objects in VEs

If a user views an object (a model or a visualization) she/he may want to move it around. One reason for doing this is the desire to look at it from different perspectives. Although in VE this can can be accomplished by simply walking around the object, it still may be desirable to move the objects instead of changing the position of the viewer.

An intuitive method to move objects in 3D space is the *model-on-a-stick* technique, which is depicted in Figure 2.21. If the user performs a pinch, the application acts like she/he grabs a stick connected to the model she/he wants to move. Using this stick she/he can move the object or rotate it about the position of her/his hand.

This is accomplished by using a coordinate system attached to the users hand. This coordinate system can be given as a matrix[4], A_{HTW}, mapping coordinates from the hand coordinate system to the world coordinate system. If the user initiates the movement by beginning the pinch

[4]Using VLIB from Fakespace Inc., Mountain View, CA, this matrix can be obtained using the function `GetTrackerMatrix()`.

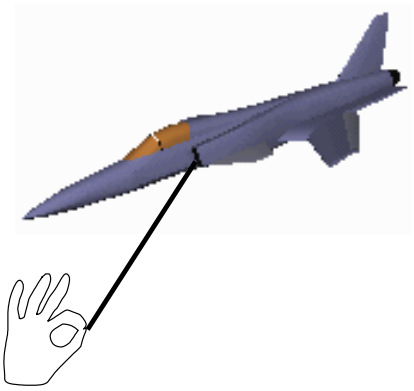


Figure 2.21: The “model-on-a-stick” technique.

gesture, the coordinates are transformed from world coordinates to hand coordinates. Instead of changing the coordinates, the transformation $A_{\mathrm{HTW}}^{-1} A_{\mathrm{objTrans}}$ (where A_{objTrans} is the current transformation used to transform the object in world coordinates) can be saved as A_{Pinch}. This is shown in Figure 2.22.

If the user moves and turns her/his hand, the current hand-to-world transformation is used to display the object described in the hand coordinate system. This is the same as displaying the original object and replacing its corresponding transformation A_{objTrans} with the transformation $A_{\mathrm{HTW}} A_{\mathrm{Pinch}}$. At each moment, the object expressed in hand coordinates is displayed with the current hand-to-world transformation as shown in Figure 2.23.

This has the desired effect of putting the object at the end of a stick, that reaches from the position of the hand to the position of the object at the begin of the pinch and moving the object with this stick when moving the hand while holding the pinch gesture.

It turns out that this technique provides an intuitive means of changing the position and orientation of objects. Users quickly become used to this technique and can learn in almost no time how to use it.

2.16 Related work

A number of techniques have been introduced to simplify scalar fields. Many of these operate on grids, *i.e.*, discrete data sets with a given connectivity. Most techniques use either a bottom-up or top-down strategy. A bottom-up strategy starts with a high-resolution representation of the data and examines it to identify regions where the grid can be simplified while introducing minimal error. This process is repeated until a given error threshold is reached. Top-down strategies start with a coarse mesh and insert points into the mesh in regions of maximum variation until a desired error condition is met.

Cignoni *et al.* [4] designed several top-down algorithms. They use a Delaunay-based procedure to generate a mesh that represents a set of points with associated scalar values. This mesh is refined by finding a data point whose associated function value is poorly approximated by

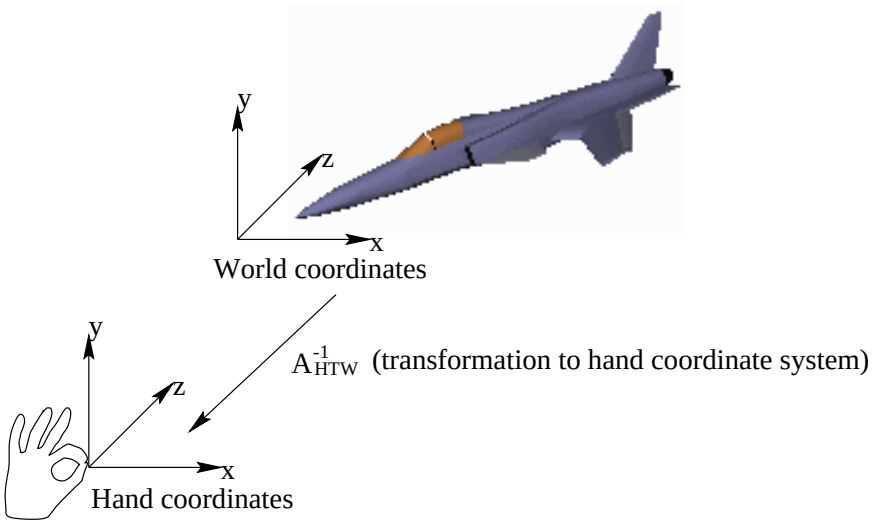


Figure 2.22: Model-on-a-stick — begin of a pinch.

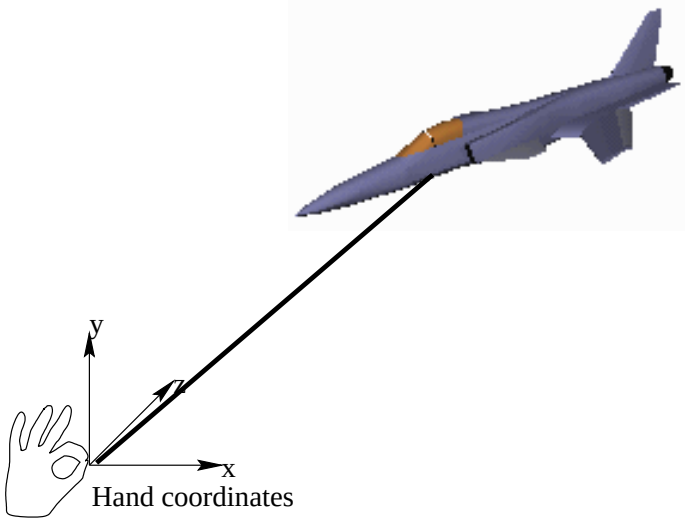


Figure 2.23: Model-on-a-stick — Dragging an object.

the mesh and adding this point to the triangulation. After adding this point, the mesh is then modified to preserve the Delaunay property. This algorithm has been generalized to include a decimation strategy [5]. This is done by collapsing edges of the triangulation. The selection of the edge to be collapsed is performed by analyzing the gradient at each vertex of the grid. It is calculated as a weighted average of gradients of all the tetrahedra sharing that vertex. The weight of a tetrahedron is given by the solid angle subtended by it at the vertex. The edges are ascendingly sorted by the difference between the gradients at the endpoints of each edge and then successively eliminated until a threshold is reached.

Kreylos *et al.* [24, 25] create a near-optimal linear approximation to scattered data in one or two variables. In the case of scattered data in two variables, a Delaunay triangulation is used to triangulate a set of points discretizing the function. The positions and connectivity of these points are changed using a simulated annealing scheme to find the optimal placement for the given amount of vertices, minimizing the error of approximation resulting from using linear interpolation on the triangulation of these vertices. Several levels of details can be generated by adding further vertices to a given approximation. Simulated annealing is used to find the optimal positions for the newly added vertices, while the preserved vertices from the previous approximation are restricted to remain at their positions.

The algorithms of the bottom-up category roughly fall into two classes: Algorithms simplify a mesh by removing simplices or utilize a wavelet transform to simplify the data.

Algorithms removing simplices include the work of Schroeder *et al.* [38]. They create approximations to the original data set by removing vertices from the original mesh. These vertices are selected by a distance-to-plane criterion. If the distance of a vertex to the average plane of its surrounding vertices is below a specified threshold, then the vertex is removed. The resulting hole in the mesh is then be re-triangulated.

Hoppe *et al.* [18] use edge-collapse operations to create a simplified version of an original mesh. A continuous-resolution representation is created by storing a coarse mesh resulting from performing edge collapse operations and a list of the edge collapse operations. Finer levels of detail can be created by countermanding the previously performed collapse operations utilizing the stored list of these operations.

Gieng *et al.* [10] remove entire triangles from a triangular mesh. They create hierarchies of meshes approximating a surface by removing individual triangles from the mesh. The triangles are collapsed to points whose location is calculated using a criterion based on the curvature of the surface. Trotts *et al.* [43] generalize this approach to scalar fields defined over tetrahedral meshes. They collapse tetrahedra by a series of edge collapse operations. Staadt and Gross [40] have developed a similar algorithm.

Nielson *et al.* [31, 32] use a wavelet approach to simplify vector fields defined over the sphere or curvilinear grids. In [31], they define a class of Haar wavelets over triangular domains and use those to simplify a vector field over a sphere. In [32], they apply a “lifted Haar wavelet” to curvilinear grids.

Chapter 3

Vector field simplification based on clustering

3.1 Approach I: Clustering vector fields using a single distance function

The first approach is a modification of clustering as applied to surface reconstruction. Instead of positions on a surface, the data points are now the samples of a vector field. Since the original surface reconstruction algorithm uses a distance measure for the data points, this must be replaced. The distance measure is the only source of information the algorithm uses about the vector field. It must contain all relevant information concerning the samples, *i.e.*, positional and vector information. An obvious choice is to use a weighted combination of all magnitudes, as mentioned in Section 2.10:

$$\lambda_1 \|\mathbf{p_1} - \mathbf{p_2}\| + \lambda_2 \angle(\mathbf{v_1}, \mathbf{v_2}) + \lambda_3 \Delta_L \quad . \tag{3.1}$$

$\mathbf{p_1}$ and $\mathbf{p_2}$ are the sample positions, and $\mathbf{v_1}$ and $\mathbf{v_2}$ are the vectors at these locations. The $\lambda_i > 0$, $i \in \{1, 2, 3\}$, are weights (that do not necessarily have to add up to 1).

Several split strategies have been examined:

1. Split in direction of maximum extension.
2. Choose the position and value of the vector with maximum deviation from the current cluster center, provided it does not have the same position as the current cluster center.
3. Split in the direction of maximum deviation, regardless if this is a positional dimension or a part of the vector.

Strategy 1 computes the extension for each spatial dimension of the cluster. This is done by computing the standard deviation of the cluster samples from the cluster center in each spatial dimension (*i.e.*, for each dimension of the position of the sample). The dimension with the

maximum deviation is chosen and two new cluster centers are created by adding an offset to the original cluster center in this dimension. Strategy 2 simply chooses the sample of the current cluster, having the maximum distance (with respect to the used distance function) to the cluster center and uses this sample as the representant of a new cluster. Strategy 3 works like strategy 1. However, it does not only take positional dimensions but also the vector information of the sample into account. Thus it is possible that two cluster centers with identical position, but different vector value are created.

The main advantage of this approach is its robustness. When a new center is not placed in the middle of a region consisting of vectors of similar length and direction, it will move toward the center of this region. The distance measure used for associating a vector with a cluster also takes the vector part into account. Thus all vectors of similar length/direction in a region will be associated with the new cluster. The new cluster center is recomputed as the weighted average of all samples. This causes the representant to lie in the middle of the region.

Unfortunately, the choice of a single distance measure for all purposes leads to several problems. In many cases it will create disjoint clusters. The final result of the clustering is to be used by either a triangulation step to create a triangular (or tetrahedral) mesh representing the vector field or for scattered data interpolation. Thus it is desirable that the clusters are "connected regions" in space. Both, position- and vector part are mixed in the distance measure that is used to determine the cluster to which a sample belongs. Thus it is possible that two regions of vectors with similar direction and length that are position-wise very close together are merged, even if they are separated by a region of vectors of different direction or length. This is illustrated in Figure 3.1. It happens mainly because combining the positional and vector information in one distance function causes the clustering to treat the samples as $(n + m)$-dimensional[1] entities, and creates $(n + m)$-dimensional coherent clusters. Looking at the clusters in the positional domain is the same as projecting them down into n-dimensional space. This projection of the $(n + m)$-dimensional clusters is not necessarily coherent in the positional dimensions. One possible solution is to choose a distance function that depends more strongly on the distance of clusters, however this increases another problem.

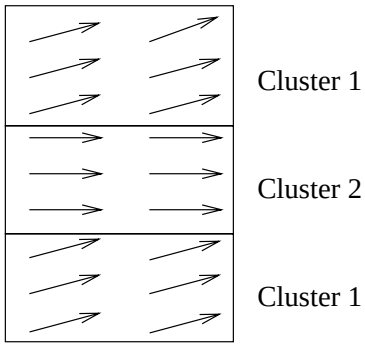


Figure 3.1: Approach I — "Disjoint clusters".

[1]Regarding the meaning of n and m see Definition 1

If a cluster contains mostly vectors that have a similar direction and length, it should not be split. Splitting a cluster should only happen, if the vectors within deviate from each other by a certain amount. Taking the positional part into account causes a penalty for big clusters that should not be present. This problem could be alleviated by weakening the influence of the positional part in the distance measure. However, this is exactly the opposite of what should be done to solve the previously discussed problem. The two problems in conjunction suggest that it is necessary to use different distance measures. One distance function should be used for deciding on what cluster to split next. A different distance function should be used to associate the samples to clusters. Splitting the clusters should mostly depend on how much the vectors deviate within a given cluster, and assigning a sample to a cluster should mainly be governed by spatial proximity.

Further problems in using this approach arise from the need to determine adjacency of clusters. An efficient implementation of the clustering algorithm needs to determine adjacency relationships between clusters. Only samples in clusters adjacent to the split clusters are marked for reclassification, *i.e.*, potential assignment to a different cluster. If the Gabriel graph is used to determine neighborhood information, then this requires finding a sample between two given samples, both position- and vector-wise. Determining the vector in the "middle" between two given vectors can prove to be a problem. Furthermore, it is desirable to replace the Gabriel graph with the more natural Delaunay graph. For the vector part this is a tedious undertaking, if a distance function different from the Euclidean norm is used[2].

The later problem can be solved by using only the position part in determining adjacency information. The end result of the clustering process should consist of coherent clusters. Thus determining neighborhood only using the position part seems to make sense. However, if this is done it is possible that sample are overlooked in the reclassification process. This is illustrated

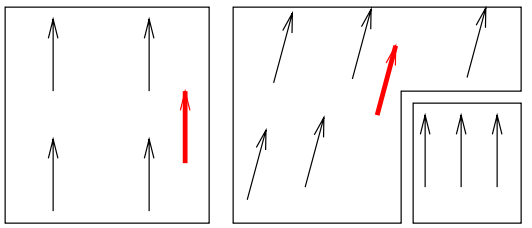


Figure 3.2: Problems arising from using only positional information to determine adjacency.

in Figure 3.2 (bold red arrows represent the cluster centers). Like Figure 3.1 it shows an example for a disjoint cluster. If the clustering algorithm were to split cluster 1 in this vector field and place the new center in the region on the right side of cluster 2, then the vectors in that region would not be placed on the reclassification list. Cluster 2 is between the new cluster and the split cluster. Thus the newly created cluster is not adjacent to the cluster it was meant to refine. This will most probably result in no vectors being assigned to this cluster. A possibility to avoid

[2]Refer to [33] for further information.

this problem is forcing the cluster that was split to be on the reclassification list. However, parts of other disjoint clusters could still be overlooked in the reclassification process.

A different approach to solving this problem is to check for each sample to be assigned to a given cluster, if it is adjacent to that cluster. This can be done by taking all adjacent clusters for this cluster into account and checking, if they are between the sample and the cluster it should be assigned to. However, even if the computational cost using the Gabriel graph to determine adjacency information might be acceptable in order to prevent disjoint clusters, using more accurate ways to determine adjacency (like the Delaunay graph) might increase the additional imposed computational cost to unacceptable levels.

Another problem seems to invalidate this approach. To determine if a sample is adjacent to a cluster, the old cluster centers (before the samples are reclassified and the centers are recomputed) are used. Thus, the algorithm might consider a sample to be adjacent to a cluster even if it is not. Checking whether a cluster is disjoint, removing the samples in disjoint regions of the cluster, and assigning them to different clusters might solve this problem. However, this would most likely require an iteration checking the clusters after change in sample assignment. The iteration needs to be performed until all clusters are coherent. This imposes further computational cost and could slow down the clustering process to unbearable levels.

Furthermore, using approach 3 to determine the location of the new center after the split process[3] leads to the possibility of centers having the same position but different vector components. This is illustrated in Figure 3.3. The two cluster centers are at the same position. This

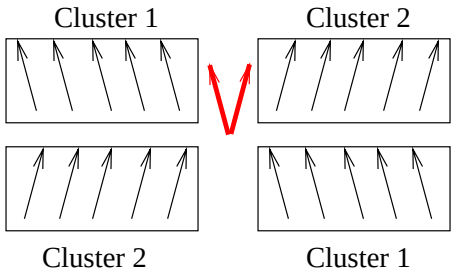


Figure 3.3: Two centers at the same position — a problem when computing the triangulation.

poses major problems, when the cluster centers are to be used as vertices in a triangulation of the vector field. Each vertex in a triangulation must be associated with an unique value. However, the cluster partition shown in Figure 3.3 assigns two distinct values to the same position. Even though this problem is intensified by splitting in vector dimensions, it even arises if the split points are only determined by changing the position part of the vector center. It is desirable that the clusters form convex regions to simplify the task of finding an actual triangulation for the vector field. Changing only the position part of the cluster center during the splitting part of the clustering process does not seem to be viable since the vector part of the new centers is used in assigning the samples to the new clusters. On the one hand this part is used in assigning vectors to the cluster. On the other hand, it is not chosen to fit the vector-direction and length in

[3] Compare new-center-placement strategy 3, described on page 36.

the region that is to be approximated more closely.

Taking all this into account, this approach dose not yield satisfactory results. Especially its tendency to split big clusters will result in a homogeneous tessellation of the vector field and not in one that is finer in regions of detail and coarser in regions of almost constant behavior.

3.2 Approach II: Clustering vector fields using different distance functions

The second idea is to use two different distance functions to achieve the desired results. This should pose an improvement over approach I because it takes the two contradicting requirements of the distance function into account:

1. The *split distance* function is used to determine which cluster is split in the next splitting operation. This function should mainly depend on the variation of the vector part.

2. The *assignment distance* function determines which cluster a sample is assigned to. It should mainly depend on the positional part. This is necessary because the desired result is spatially coherent clusters. This is necessary to avoid situations like those depicted in Figures 3.1, 3.2, and 3.3. They result in problems when using the cluster centers as input for a triangulation scheme or a scattered data interpolation method.

This leads to an approach that uses a split distance function only dependent on the vector part of samples:

$$\lambda_1 \angle (\mathbf{v_1}, \mathbf{v_2}) + \lambda_2 \Delta_L \quad .$$

$\mathbf{v_1}$ and $\mathbf{v_2}$ are the vectors of the samples. The $\lambda_i > 0$, $i \in \{1, 2\}$, are weights (that do not necessarily have to sum up to 1). As assignment distance function the Euclidean distance is chosen which is only dependent on the position of the sample. This yields clusters similar in shape to the cells in a Voronoi tessellation that uses the cluster centers as positions of the sites. The clusters do not exactly correspond to the Voronoi cells. As soon as the location of the representants, or centers is chosen, all points are reclassified, *i.e.*, assigned to the closest cluster. The cluster representants are subsequently recomputed as the centers of those samples. This will change their position and causes that clusters are not exactly shaped like Voronoi cells. Furthermore, the cluster shape is governed by the positions of the samples. This will also prevent the clusters from being shaped like Voronoi cells.

Based on a split distance function only dependent on the vector component of the samples there are several ways to compute the error for a given cluster. In the following definitions, a cluster C is considered to be a set containing all samples assigned to it as members. $|C|$ denotes the cardinality of the set, *i.e.*, the number of samples in cluster C. A sample s_i represents a tuple of position and vector in that position, $s_i = (\mathbf{p_i}, \mathbf{v_i})$. The deviation of a sample s from the cluster center, given by the split distance function, is denoted by ε_s.

1. The *error sum* is the sum of all individual deviations of the samples form the cluster center, called

$$\varepsilon_{C,\text{sum}} = \sum_{s \in C} \varepsilon_s \quad . \tag{3.2}$$

2. The *mean error* is the error sum divided by the number of samples in the cluster, called

$$\varepsilon_{C,\text{mean}} = \frac{1}{|C|} \sum_{s \in C} \varepsilon_s \quad . \tag{3.3}$$

3. The *max error* is the maximum of all individual deviations of the samples from the cluster center, called

$$\varepsilon_{C,\text{max}} = \max\{s \in C | \varepsilon_s\} \quad . \tag{3.4}$$

Method 1 places a penalty on large clusters and should be avoided. Using these definitions, it is possible to define the sum, mean, and maximum error of a cluster representation of a vector field analogously, as the sum, mean, or maximum of the individual cluster errors ε_C.

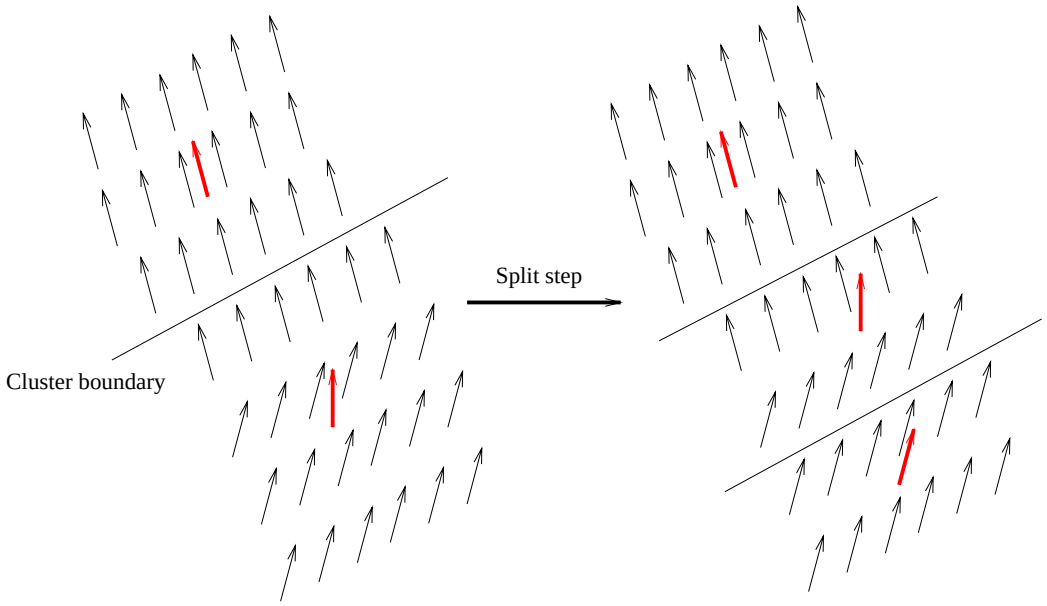


Figure 3.4: Increasing error with new center.

Unfortunately, this approach is not as tolerant when it comes to choosing an unsuitable location for a new center. This is due to the fact that only the position is taken into account when

assigning a sample to a cluster. Figure 3.4 (red, bold arrows denote the cluster centers) illustrates this problem. The left-hand side shows two clusters representing a vector field. Moving each representant into the center of the region consisting of vectors similar to it would improve the quality of the approximation. If the vector part played a role in determining to which cluster a sample should be assigned, this would happen automatically. The vectors similar to the cluster representant vector would be assigned to each cluster, thus generating better cluster boundaries. Recomputing the cluster representant as average of the positions of all samples belonging to a cluster would move the centers toward better positions. However, if only the position parts are taken into account this correction of poorly chosen centers does not take place. Instead, a new representant might be inserted between the two existing representants. This representant collects the nearest samples around it. The resulting cluster has a larger mean error than the mean error of the cluster prior to the splitting process. If the next cluster centers are also poorly chosen, the approach will try to refine this region. More clusters are created in the vicinity and thus other regions are neglected. Moving poorly chosen representants to a better position could avoid this situation.

This approach also restricts the clusters to have shapes similar to the cells in a Voronoi tessellation. A positive aspect of this fact is that it ensures the clusters to form convex regions. This helps later during the triangulation step. One problem with clusters shaped like Voronoi cells is illustrated in Figure 3.5. In some cases (edges marked with an “G” in Figure 3.5) the Voronoi cell boundary (dashed line) intersects the dual dual Delaunay edge (solid line) exactly in the midpoint. If linear interpolation is used, this marks exactly the point, where the contribution of one cluster center gets larger than the contribution of the other cluster center. This happens exactly at the cluster boundary which is exactly the point, where it is supposed to happen. The contribution of a cluster center should be larger within the cluster region than the contribution of other cluster centers. Unfortunately, there are cases where the Delaunay edge does not intersect its dual Voronoi edge. Instead it intersects other Voronoi edges. In this case the edge is partially in a cluster whose center does not contribute to the interpolation value at all. Thus the edge is in the wrong cluster and assigns values to points within the cluster without taking the actual cluster center value into account. This may cause problems when using a Delaunay triangulation of the cluster centers to represent the vector field and should be taken into account, when the final triangulation step is devised. Note however, that this is a problem of the Delaunay triangulation in general. Nonetheless, in general this approach yields promising results. Finding solutions to its shortcomings should be easier than dealing with the non-coherent clusters resulting from approach I. However, this approach should be modified to increase its robustness in regard to choosing a poor position for the centers during the split process. It may be possible to achieve this goal by creating a hybrid of this approach and approach I.

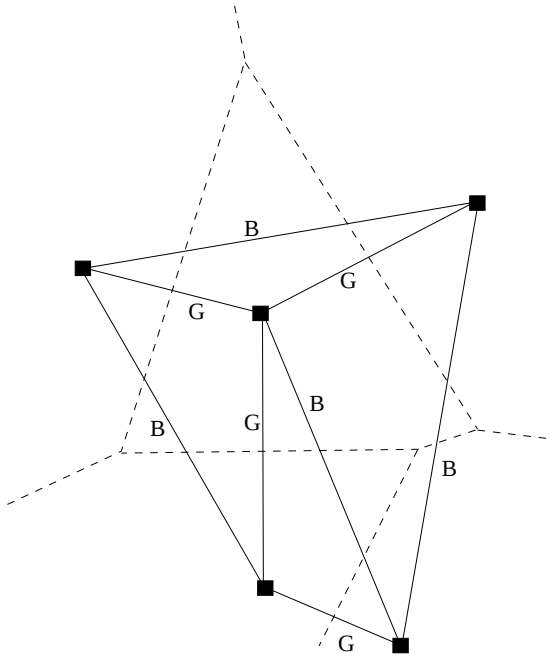


Figure 3.5: Problems using Delaunay triangulation

3.3 Approach III: Clustering vector fields using a two-pass approach

During the examination of how well the clustering approach could be adapted to vector fields the idea of using two distinct passes came up. One pass performs clustering only according to the vector part of the samples, and a second pass only takes the position of the samples into account. For the distance function between two positions, the Euclidean distance is an obvious choice. For the vector part it is more difficult to find a distance function. It is necessary to choose a sufficiently powerful, *i.e.*, customizable, distance function, and to generalize the Gabriel graph to this distance function. In order to compute the Gabriel graph for positions it is necessary to compute the mid point between the positions, *i.e.*, the point that lies on the line segment connecting the two positions and that has the same distance to both positions. Doing the same for vectors proves to be more complicated. For this reason, a first implementation of the two-pass approach used the Euclidean distance for both, vector part and position of the samples. This ensures that the mid-vector can be computed in the same way as the mid-position is computed. A refinement of this implementation uses the same distance measure for the vector part as approach II. Computing the mid-vector is only implemented partially because of ambiguities. These ambiguities arise when the vectors point in opposite directions and have the same magnitude, *i.e.*, they sum up to the null vector $\mathbf{0}$. Figure 3.6 illustrates this problem. In this case it is not possible to determine the correct mid-vector. The two possible choices are drawn as dashed arrows in Figure 3.6. It is not possible to choose one of these two vectors over

the other. If the distance function is changed, then the computation for the mid-vector must

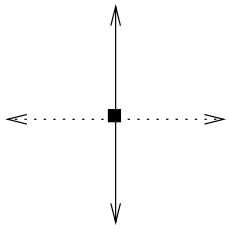


Figure 3.6: Ambiguity computing the mid-vector.

be changed accordingly. Even the generalization to the distance function used in approach II was not done completely because of the mentioned problems. Other problems arise when the Delaunay graph is used to determine adjacency information. Generalizing the Delaunay graph for general distance functions is even more tedious (see [33]).

Besides this, clustering vector fields using this approach causes more problems. These result from the fact that two passes are used in the clustering process. Since the first pass clusters only according to the vector part of the samples, the positions of the resulting clusters representants have no meaning. Spatial distance is of no importance in determining cluster affiliation of samples during the first pass. Samples having a similar vector part are assigned to one cluster regardless of whether they are separated by regions of samples with different vector parts. Since the center is the mean of all positions of samples in a given cluster its position is meaningless.

This is the reason for performing a second, position-dependent pass after the first pass. In this pass, clusters of samples with similar vector parts are split into clusters that are spatially coherent. This poses a major problem. Since no adjacency information for the samples is given, the resolution of the final clustering must be chosen by imposing a maximal positional error. In order to detect that two regions belonging to one cluster are separated by a region belonging to another cluster, the maximum size for the clusters in the position pass must be chosen sufficiently small. This causes all clusters to be of approximately this size. Even if most of the clusters could be approximated by larger regions, it is necessary to restrict the cluster size. Otherwise, spatially disjoint clusters can emerge. Some kind of merging step, coalescing adjacent small clusters that originated from one cluster of the first pass, would be necessary to countermand this problem. However, the better the resolution and the smaller the clusters resulting from the second pass the higher becomes the computational cost of this step. This step is further complicated by the restriction that the resulting clusters should form “coherent regions” in space.

It may be possible to use this approach in cases when connectivity information is given (see Section 6.2) but due to the mentioned problems this approach seems to be unsuited for scattered data without connectivity.

3.4 Comparison of the three approaches for clustering vector fields

Taking all this into account, approach II, *i.e.*, using two different distance measures in one pass, seems to be the best. However, it still needs refinement. It is desirable to incorporate the robustness of the first approach in regard to choosing an unsuitable center in the splitting process into this approach.

3.5 Refining the approach using different distance functions

One possibility to create a synthesis of approach I and II is the following:

1. Split a cluster, *e.g.*, by choosing the new center at the position of the sample whose vector part has the largest deviation from the center vector part.
2. Check all surrounding clusters for vectors similar to the new cluster center. Assign these vectors to the new cluster. The adjacent clusters are given by the Gabriel or Delaunay graph of the cluster representants.

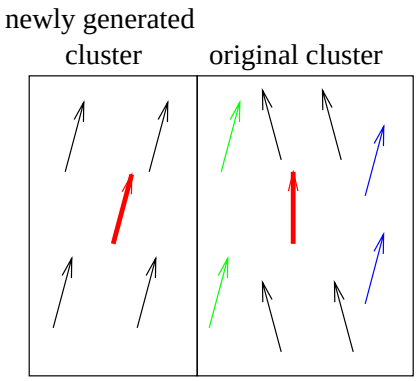


Figure 3.7: A hybrid method: combining approach I and II.

While reassigning samples to clusters depending only on the vector part it is necessary to be careful to avoid the problems of disjoint clusters mentioned during the discussion of approach I. Since the position of the samples is completely disregarded, the danger arises that samples from the “opposite” side of an adjacent cluster are assigned to the newly created cluster. The “opposite” side refers to the side from the center of the neighboring cluster that is is not adjacent to the newly generated cluster. This is illustrated in Figure 3.7 (the cluster centers are drawn as bold red arrows). It shows a newly generated cluster and one of its neighbors. The two samples drawn as green arrows should be assigned to the new cluster to move its center in the correct direction. Two other samples, drawn as blue arrows, must not be assigned to this cluster, since they are on the opposite side and would move the center of the newly created cluster too far

into the direction of this cluster. This situation can be avoided when a scalar product based criterion is used. When a sample of a cluster adjacent to the newly created cluster is to be reassigned to the new cluster, a simple computation is performed. First, the vector pointing from the representant of the cluster containing the sample to the sample position is computed. After that, the vector pointing from the representant of the cluster containing the sample to the newly created cluster is computed. Finally, the scalar product of these two vectors is computed. If it is greater than or equal to zero, then the sample may be safely added to the new cluster. However, using the scalar product this way allows vectors with directions in a range of 180° to be assigned to the new cluster for purposes of computing the center of the new cluster. This causes new problem cases. One of these cases is illustrated in Figure 3.8. We assume that either the upper or lower vector on the right side is chosen as vector with maximum deviation (both can be chosen since they both have the same direction and length). Furthermore, we assume that no vector in an adjacent cluster is added to the new cluster. In that case those two vectors are sorted into the new cluster and the center is placed between them in a region with vectors of different length/direction. This places the new center in a different region than the the sample with the maximum vector part deviation from the cluster center.

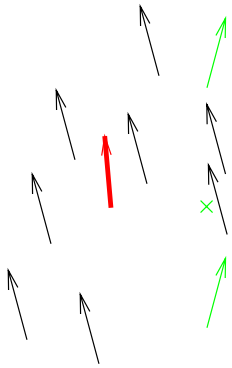


Figure 3.8: Problems computing the center of the new cluster — Case 1.

The case depicted in Figure 3.9 is even worse. Regardless which of the four samples in the middle region (drawn as green arrows) is chosen as maximum-deviation sample, two others are added to the cluster as well. This results in placing the center of the new cluster center at the same position as the old cluster center. This will result in either one cluster being empty (if all samples are assigned to the first cluster of different clusters with equal spatial distance to the sample) or in a non-deterministic splitting of the cluster (possibly resulting in disjoint clusters). Both cases are unacceptable and should be avoided. Even if one would test if the scalar product is greater than zero, the situation will arise in which the new cluster-center is located extremely close to the old cluster center. Since the cluster boundaries roughly conform to the boundaries of the Voronoi tessellation using the cluster centers, this will result in empty Voronoi cells and thus in empty clusters.

Even if the optimal location for a cluster center is found, problems arise. Figure 3.10 shows an example with two clusters. If one of these clusters is split, *e.g.*, the lower one, and the center

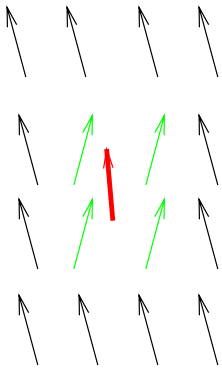


Figure 3.9: Problems computing the center of the new cluster — Case 2.

for the new cluster is computed using the four samples drawn in green, it is placed at the ideal location. For the final association of samples to clusters the Euclidean distance is used. Thus, even if the cluster consisting of the green samples is an ideal representant for the vector field in that region, a cluster consisting of these samples is not possible. The representant of this cluster would have a bigger Euclidean distance from the samples that are to be assigned to this cluster than the two original cluster representants. This results in the samples being assigned to their original clusters and the new cluster being empty. Thus it provides no improvement for the representation of the vector field. Apart from adding a new empty cluster, this split process has no effect. The same cluster will be chosen as split candidate again, yielding an infinite loop. Using the Delaunay triangulation intensifies this problem, since now the complete neighborhood is considered. The Gabriel graph only considered a subset of the neighboring clusters (see Section 2.5). In fact this problem was reveled in its whole severity, when the implementation was changed to use the Delaunay graph instead of the Gabriel graph.

This in mind, another attempt was undertaken to make approach II more adaptable in regard of disadvantageous placement of a new cluster representant. The attempt was made to reach this goal by modifying the assignment distance function. Instead of using only the position of the samples for this distance function, the combination of all criteria discussed in Section 2.10 (as shown in equation 3.1) is used. The weighting factors are chosen in a way, that the positional weighting factor λ_1 is large, and the weighting factors for the vector-related distances λ_2 and λ_3 are small.

At first, the implementation exhibited the same problems as the approach using only one general-purpose norm. Closer examination revealed that the use of the exponential function $1 - \mathrm{e}^{-l}$ to normalize the positional distance to lie in the interval between 0 and 1 causes these problems. If the vector field is defined over a larger region and the distances between the positions of the samples become larger, then the normalized value becomes so small, that choosing a large value λ_1 cannot counterbalance this effect. Despite of the goal to put an emphasis on the position of the sample, the use of the exponential function puts an emphasis on the distance between the vectors associated with the samples.

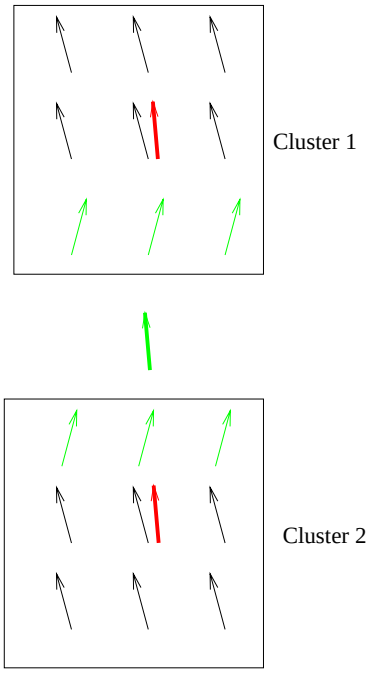


Figure 3.10: Problem arising from using two distance functions

It is possible to prevent this by replacing the exponential function with a different means for normalizing the distance between the positions of the samples. One way to accomplish this is to compute the bounding box of the vector field and divide the components of the position by the extent of the bounding box in that particular dimension. This yields better results. But further experiments have shown that the results of this approach cause problems when used to generate a triangulation. This is illustrated in Figure 3.11. Because the assignment distance function takes the vector distance between samples into account (even with a small weight), this permits a greater variety of possible cluster shapes, *e.g.*, clusters like Cluster 4 in Figure 3.11. This cluster extends widely into one dimension but is narrow in the other dimension. This is due to the fact that the influence of the vector distance function causes it to resemble the region of vectors that are similar to its representant. In that way, the clustering algorithm produces exactly the same result, as a human might intuitively expect and generate when asked to perform the same task. This illustrates that the intuitively expected result might not be the ideal input for a triangulation or scattered data interpolation scheme. Using this approach, the interpolation scheme has to be modified, as discussed in Section 6.2. From this point, Heckel has extended the work on clustering. The results are discussed in [16].

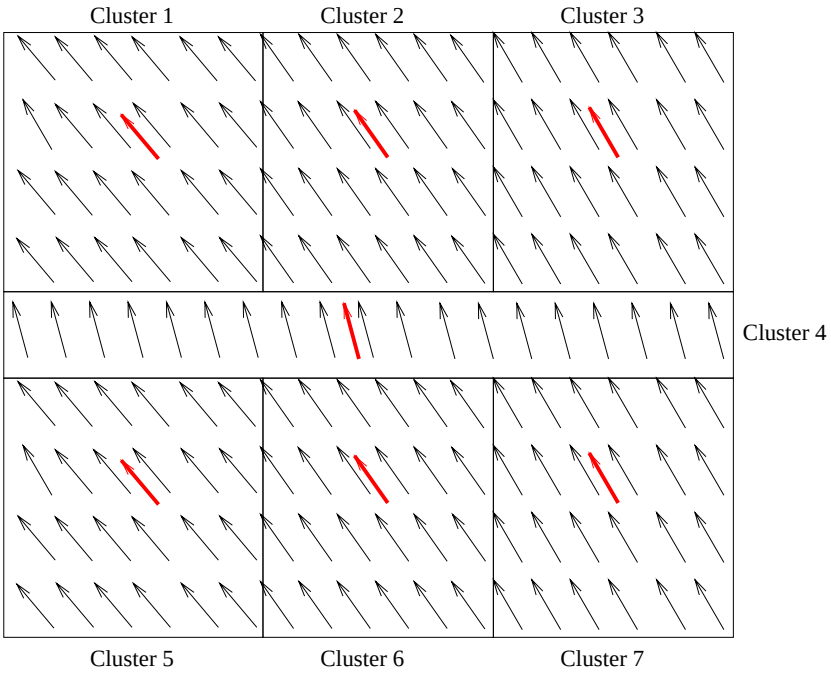


Figure 3.11: Problem arising from a wider variety of possible cluster shapes.

Chapter 4

Visualization

4.1 Utilizing a scattered data vector field hierarchy for visualization

The clustering process discussed in the previous section yields a hierarchical representation of a vector field described without point/sample connectivity. This representation can be viewed as a binary tree, in which each node has either no or two successors. Each node represents a cluster that existed at any time during the generation of the hierarchy. Each node corresponds to a region in the vector field. This region is the convex hull of the samples of the original data set that were assigned to the cluster corresponding to this node. If a finer representation is available, then the node has two descendants, which correspond to the two clusters resulting from splitting the cluster corresponding to the current node. In the further course of this work, this tree will be called *clustering tree.*

Definition 24 (clustering tree)
The clustering tree is a binary tree corresponding to the evolution of the clustering process. Each node in this tree corresponds to a cluster that existed at a given point of time during the clustering process. Each node in the clustering tree has either no or two descendants. Each node contains the following information:

1. *the sample* $s_{\mathrm{node}} = (\mathbf{x_{node}}, \mathbf{v_{node}})$ *which is the representant of the cluster corresponding to this node*

2. *the error describing by how much the vector field deviates from the representant in the region described by the node*

3. *a split number that indicates when the cluster corresponding to this node was split*

Using the split number, the exact progression of the clustering process can be reproduced, and any desired level of detail within the limits of the given cluster hierarchy can be attained. If the vector field is to be represented by a given number N_{Samples} of samples, then this representation can be constructed from the clustering tree by performing the first $N_{\mathrm{Samples}} - 1$ split

operations. Figure 4.1 shows the clustering tree for a simple example. The nodes contain the representant (first row), the split number (second row), and the error for this node (last row). In this representation, any level of detail that can be constructed from the hierarchy is represented by a thread connecting the nodes belonging to that level of detail. The clustering tree depicted in Figure 4.1 can be used to construct five different level of details. The simplest case is a thread containing only the root and thus consisting of only one representant. The next three possible level of detail threads are shown in green, red, and blue and consist of resp. two, three, and four nodes. The finest possible representation is the thread connecting only the leafs of this tree. Each thread can be constructed from the previous thread by replacing the node with the lowest split number along the thread with its two descendants. By doing this, all split operations performed during the clustering are sequentially considered.

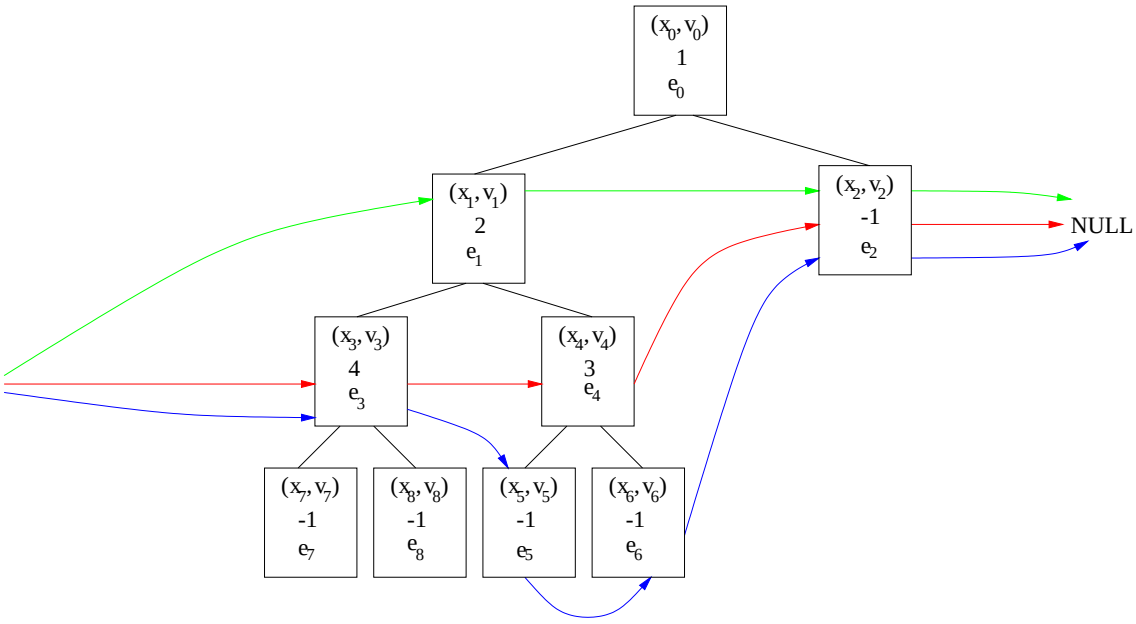


Figure 4.1: Clustering tree and thread representing a level of detail.

Because of the sequentiality of the splits, it is possible to speed up the process of finding the next split candidate by keeping an array of pointers. For each possible split number the array contains a pointer to the corresponding node in the clustering tree. This is shown in Figure 4.2. By working sequentially through this array, it is possible to reconstruct the clustering process and access every possible level of detail. This is shown in Figures 4.3 and 4.4. Figure 4.3 shows the clustering tree and the thread representing the level of detail resulting from using all pre-computed split operations up to `lastUsedSplit`$= 1$. The next split operation to use is split number two. The arrow corresponding to the node pointer in the split list is drawn in red. The result from using the pre-computed split operation is shown in Figure 4.4. The thread denoting the previous level of detail is still shown as dotted line. The thread corresponding to the level of detail after using the next split operation results from replacing the node indicated

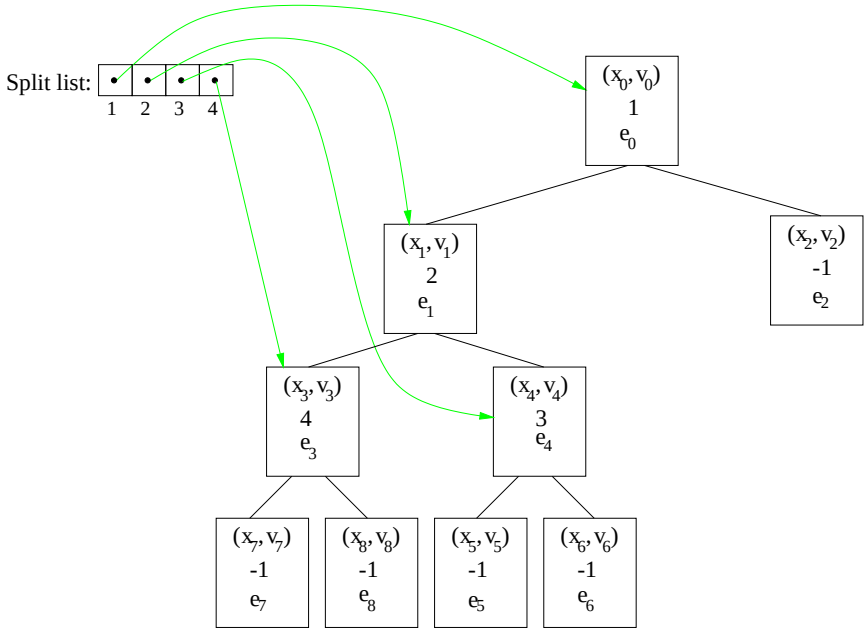


Figure 4.2: Clustering tree and array mapping split numbers to nodes.

by the pointer in the split list by its two children. After that the variable "`lastUsedSplit`" is updated, and the hierarchy level can be obtained by traversing the nodes along this thread.

A method to utilize the resulting list of representants to find an interpolation vector value for a given location within the vector field is using it as input for the localized Hardy interpolation described in Section 2.3. This method has been implemented to compare the simplified vector field to the original vector field. The implementation uses a slider to select a level of the hierarchy. The thread corresponding to this level of detail is then computed according to the described algorithm. This representation is used for the Hardy interpolation. The implementation allows to view stream lines computed from the current hierarchy level and the original vector field. Using toggle buttons, it is possible to select the resulting stream lines and difference surface connecting them to visualize the differences. It is also possible to display stream surfaces. The implementation uses *OpenInventor* to display the visualization objects. Re-positioning the stream lines and stream surfaces is possible by using OpenInventor draggers.

4.2 Tetrahedral hierarchies

The original intent of this work was to compute hierarchies of tetrahedral meshes. The clustering was to be used to obtain different representations of the vector field. These results were to be used to generate pre-computed tetrahedral meshes stored in a way enabling access to the different hierarchy levels by standard visualization algorithms. The resulting hierarchy would not have contained all possible resolution levels, but rather a sensible selection of hierarchy levels (*e.g.*, each level having an error less than a given error bound with a given list of descending

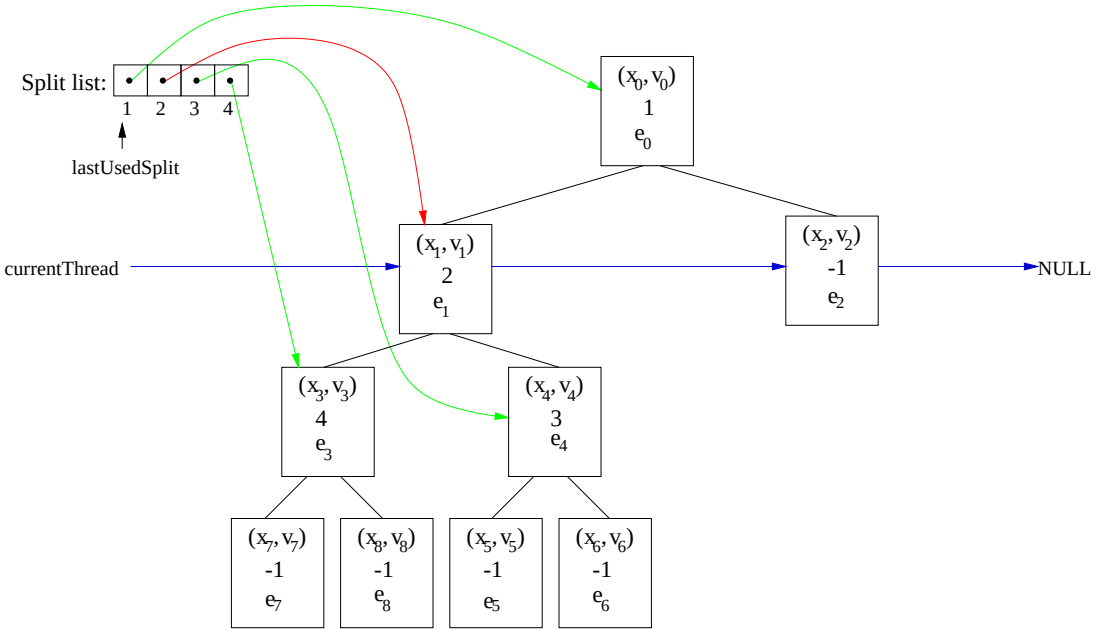


Figure 4.3: Refining a thread in the clustering tree — original thread.

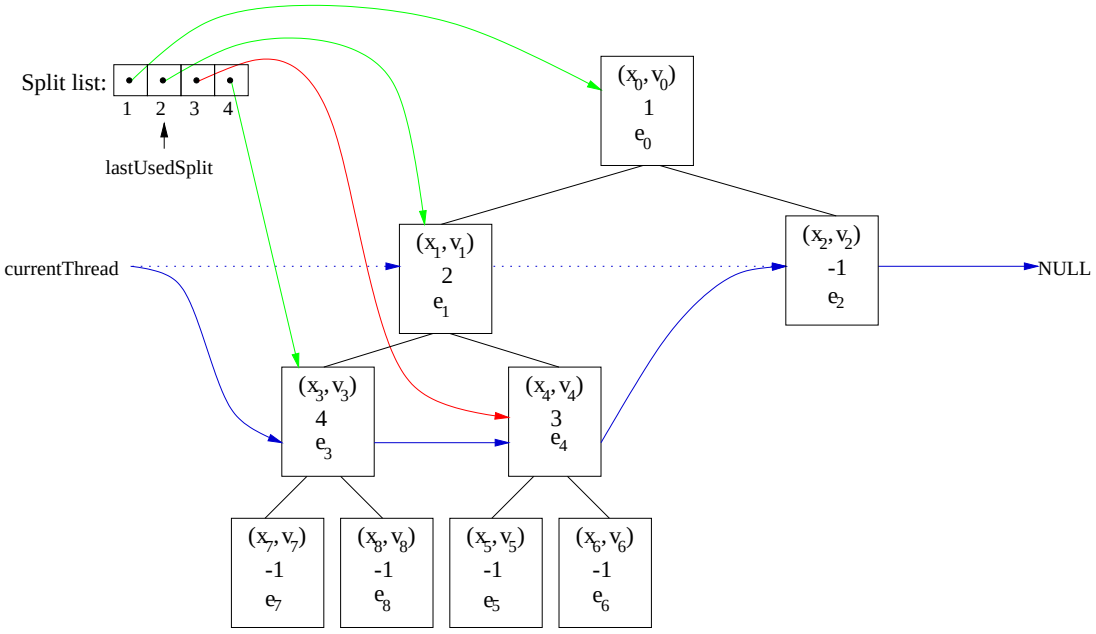


Figure 4.4: Refining a thread in the clustering tree — refined thread.

error limits). The initial implementation operated accordingly. However, it became obvious that even for a relatively small number of scattered data positions (in the range between 100000 and 1000000 sample positions) the time to compute the Delaunay triangulation becomes prohibitively high. The aim of this work is to devise methods for visualizing large-scale data sets. These contain amounts of samples that exceed these numbers by several orders of magnitude. It is likely that even a reduced and simplified version of these vector fields will contain more than a million data items. Even if triangulation or grid generation algorithms with less computational cost are used, it does not seem to be feasible to compute the tetrahedral meshes in advance.

The localized Hardy interpolation method presented in Section 2.3 circumvents this problem by not requiring any mesh. If no geometry or connectivity information has to be taken into account this method yields excellent results. Furthermore, it avoids some problems that arise when mesh-based interpolation schemes are used. Figure 4.5 shows a case where the interpola-

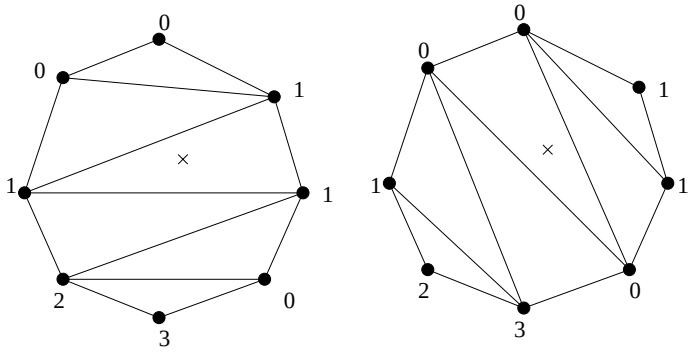


Figure 4.5: Ambiguity in approximation value due to non-uniqueness of triangulation.

tion value is not uniquely defined. This case arises, even though Delaunay triangulation is used, since the given points are co-spherical and thus no unique Delaunay triangulation exists. The two triangulations shown in this figure yield two different values for the queried point (marked with an "x"). If the used triangulation is not a Delaunay triangulation, then there are even more possibilities. The most commonly used linear interpolation within the simplices produces additional problems, *e.g.*, in many cases the resulting flow is not mass-preserving, even in many cases, like wind tunnel data, where this should be true. This problem is further discussed in [22][1].

Hardy interpolation provides an efficient means for approximation avoiding some shortcomings of grid-based approaches. However, using Hardy interpolation also causes some problems, mainly the problem of the missing connectivity, discussed at the end of Section 2.3. Additionally, there are visualization methods that require a grid, *e.g.*, the alternative method for stream line and stream surface generation presented in [22, 23]. In cases like these, it becomes necessary to generate some kind of grid for the data.

[1] It remains to be verified whether the Hardy interpolation is mass-preserving.

4.3 Procedural generation of tetrahedral meshes during visualization

An observation leading to a possible solution is the fact that in those cases when a finer representation is used only a limited region of a vector field is of interest to the user of the visualization application. As mentioned in the introduction, the need for a hierarchy resulted from the need to provide the user with a more accurate representation of the vector field, when she/he focuses on a certain region of the vector field, *e.g.*, when zooming into a region. This suggests that it is not necessary to compute a complete tetrahedrazation for the finer levels of details. It is sufficient to compute the tetrahedrazation for that part of the vector field, that currently is of interest to the user.

If the number of points that are to be triangulated is sufficiently small, then it becomes possible to generate the corresponding mesh on the fly. This allows the procedural or "on-the-fly" generation of the grid while the visualization is in progress and adapt the mesh to the current requirements. This work examines three approaches for the procedural generation of a tetrahedral mesh:

1. Generate a separate mesh for each interpolated value.
2. Generate an initial mesh around the query point and generate a new mesh each time a query lies in the margin of or outside the vector field.
3. Require the user to choose a region of interest (ROI) and generate a triangulation only for this region.

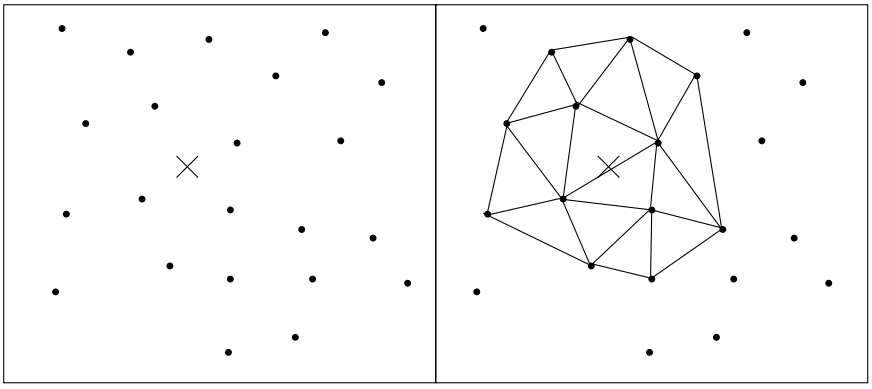


Figure 4.6: Computing a triangulation per query — first query.

Method 1 generates a tetrahedral mesh whenever a value for a given point is queried. This is accomplished by organizing the sample locations in a k-D-tree. This is the same k-D-tree used for the localized Hardy interpolation. This method is illustrated in Figures 4.6 and 4.7. The set of samples is stored without any connectivity (see left-hand side of Figure 4.6). If the

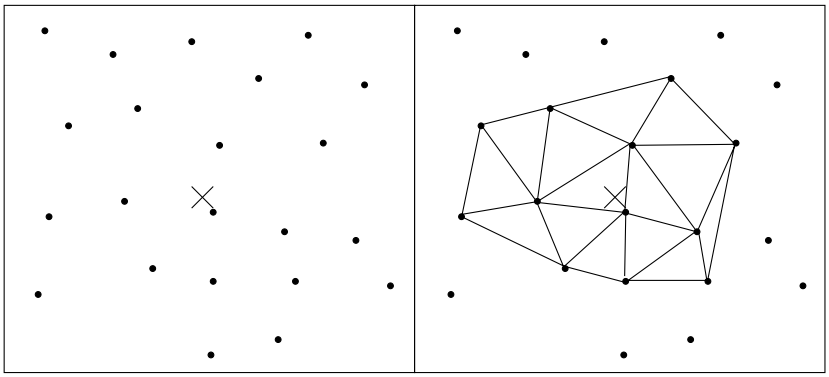


Figure 4.7: Computing a triangulation per query — second query.

value for any location is queried, then the k nearest samples are selected. The number k is a parameter that has to be chosen by the programmer or user (in Figures 4.6 and 4.7 $k = 12$). For the selected positions a Delaunay triangulation is generated which is used to compute the interpolation value for the given location. Whenever a new query is performed, the data set is again considered as simply consisting of samples (see left-hand side of Figure 4.7) and a new triangulation is computed.

It is inefficient to generate a triangulation for each query. First of all, consecutive queries are probable to lie in neighboring tetrahedra (or even the same tetrahedron). This is, *e.g.*, the case, when a stream line is computed and the time step size is sufficiently small. Successive queries to compute the Runge-Kutta solution for the differential equation will most probably be within neighboring tetrahedra. Using a pre-computed tetrahedral mesh this can be utilized by starting the search for the tetrahedron containing the queried point that is described in Section 2.6 in the tetrahedron that contained the previously queried position. If each value query generates a new grid, then it is not possible to use this simple speed up technique. Furthermore, methods using the actual grid and not only queried values cause problems when using this approach.

Each query generates a local triangulation around the original query point. As mentioned above, in many cases this grid is likely to contain the next queried point. However, using this grid for computing the next interpolation value is not advisable in all cases. The triangulation shown in Figures 4.6 and 4.7 behaves well in vicinity to the original query point. At the margin of the triangulation one typically finds a set of long, skinny triangles. This is caused by ignoring all other positions in the data set. If they were added, then the Delaunay criterion would no longer be satisfied, since several of these points lie in the circumsphere of the outer triangles. If a value for a point within those outer triangles is queried a new triangulation should be used.

Considering this fact leads one to refine the original algorithm (see 2). The first query for a value generates a tetrahedral mesh like in the original approach. However, differing from this, the mesh is stored for use in subsequent queries. To detect queries that lie in outer simplices of the mesh, these simplices are marked (colored red in the Figure). A second query, lying within the mesh in a not-marked simplex, can use the existing mesh to compute interpolation value. This case is depicted in Figure 4.9. Queries outside the grid or within the marked simplices,

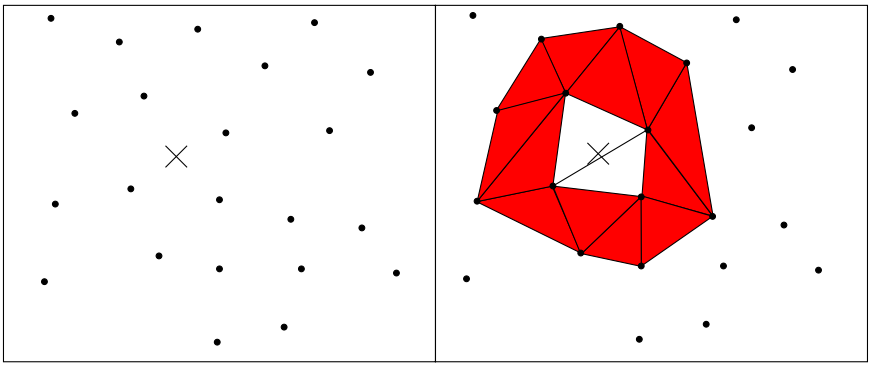


Figure 4.8: Using a triangulation for subsequent queries — first query.

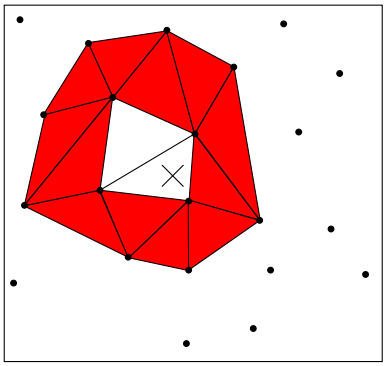


Figure 4.9: Using a triangulation for subsequent queries — second query.

like those in Figure 4.10, are considered to be invalid for the existing mesh and trigger the computation of a new mesh.

One important design issue of this version of the procedural triangulation is the size of the "buffer zone", *i.e.*, which simplices are marked to trigger a new computation of the mesh if they contain a query point.

Definition 25 (border simplex)
A simplex in a simplical mesh is a border simplex if and only if one of its vertices lies on the boundary of the convex hull of the mesh.

It is obvious that at least all border simplices should be marked. To avoid the explicit computation of the convex hull, the following observation helps to find a criterion for marking simplices that can be implemented more efficiently. This is described for tetrahedra, since the implementation for this Diplomarbeit assumes tetrahedral grids.

A tetrahedron is certain to be a border tetrahedron, if at least one its faces lies on the convex hull, *i.e.*, there is no neighboring tetrahedron for that face. However, this criterion is not sufficient to mark all tetrahedra which lie on the hull. There are tetrahedra which only have

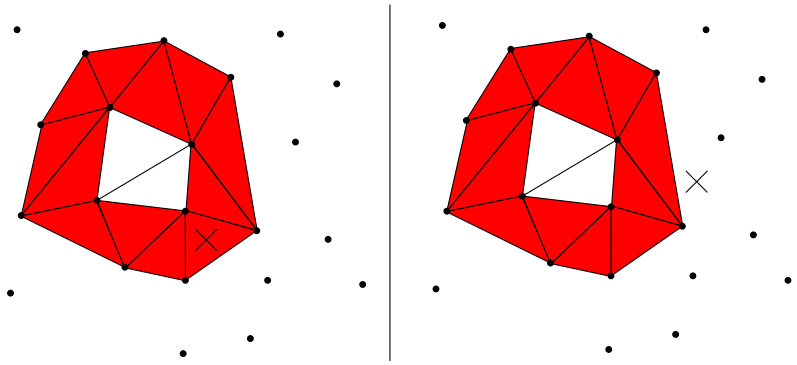


Figure 4.10: Using a triangulation for subsequent queries — invalid queries.

an edge or a vertex on the convex hull. In theses cases, they have either two (edge on convex hull) or three (vertex on hull) neighboring tetrahedra, that lie on the boundary. An efficient implementation can determine if a tetrahedron is a border tetrahedron by first checking if it has a face that is on the convex hull. This can be done efficiently, since the implementation of the "walking"-algorithm, that is used to locate the tetrahedron containing the queried point, requires a list of the neighbors for the current tetrahedron. If one face is on the hull, then the tetrahedron is marked as border tetrahedron. Otherwise, it is checked how many neighboring tetrahedra have faces on the boundary of the convex hull. The tetrahedron is marked, if at least two of its neighbors have faces on the convex hull.

The implementation for this work used only border tetrahedra to trigger the computation of a new mesh. However, even with this refinement the algorithm has shortcomings. First of all, there are applications, where the actual mesh is of interest and not its use for interpolation. If such a case arises it is possible to generate a mesh by an initial query at a seed location and generate a new grid, whenever the application uses a marked tetrahedron. Another problem with the approach is that in order for a mesh to be usable for a significant amount of queries, a sufficiently high number of samples close to the query position has to be used in the triangulation. This causes the re-computation of the the new mesh to take longer. It is hard to find the correct balance between these two demands. Additionally, the combination of this approach with a hierarchy is not straightforward. It is hard to determine, which level of detail the approach should use at a given moment. A possibility is to have the user choose the hierarchy level and work on this level, but this requires the user to be aware of the underlying hierarchy.

Because of these reasons a third approach (see 3) has been developed. This time, the user has to explicitly choose the region of the vector field that is currently of interest to her/him (drawn in green in Figure 4.11). The user does not have to be aware of the hierarchy to choose this region[2], so the approach is completely transparent in regard to the hierarchy. After the user has chosen her/his region of interest (ROI), the samples within that region are triangulated. This is shown in Figure 4.12.

Note that the triangulation does not completely "fill" the ROI. This could be solved by taking

[2]Even though it is possible to inform her/him about the currently attained accuracy.

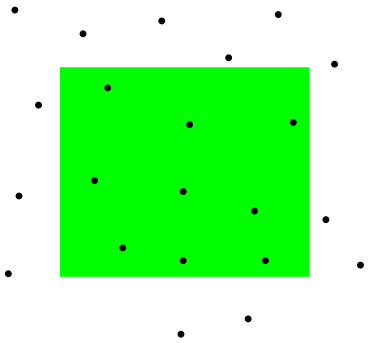


Figure 4.11: Choosing a region of interest (ROI).

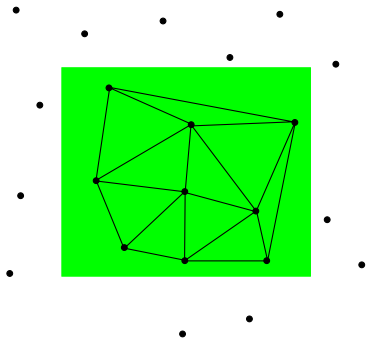


Figure 4.12: Triangulation of points within ROI.

additional samples outside the ROI into account, but doing this is not trivial. It is not possible to fill the ROI completely in all cases, even when samples outside this region are used. These cases arises, if the ROI is not completely contained within the vector field. The second problem is that it is computationally expensive to check whether the bounding box is completely within the vector field. Each of its eight vertices must be tested for being contained in the triangulation. If this test fails additional samples have to be added to the triangulation and the test for the eight vertices has to be performed again. This must continue until all vertices are included in the the triangulation. Furthermore, it also is expensive to check, whether it is possible to completely fill the ROI with a triangulation. Doing this involves computing the convex hull of the original data set and intersecting it with the bounding box. Alternatively, it is possible to use an artifice to avoid these costly computations and special cases. By interpolating the values at the vertices of the ROI, *e.g.*, using Hardy interpolation, and adding these to the triangulation, as shown in Figure 4.13, it is possible to generate a triangulation filling the complete ROI. Adding the vertices to the triangulation causes the convex hull of the data set (now containing the added vertices) and the ROI to coincide. Since the boundary of the triangulation is the convex hull of the triangulated points, as mentioned in Section 2.5, the triangulated region and the ROI coincide as well.

After the ROI is triangulated it is possible to use this triangulation to interpolate values

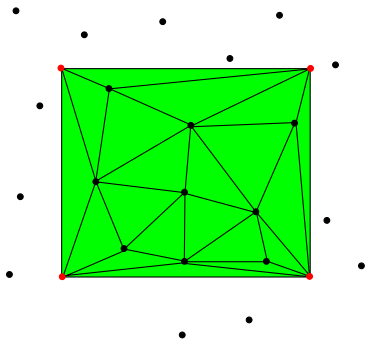


Figure 4.13: Adding the vertices of the ROI to the triangulation completely fills the ROI with triangles.

within that region, as shown in Figure 4.14. Furthermore, it is possible to use this triangulation for other algorithms that need a tetrahedral mesh, *e.g.*, the stream line generation algorithm discussed in [23].

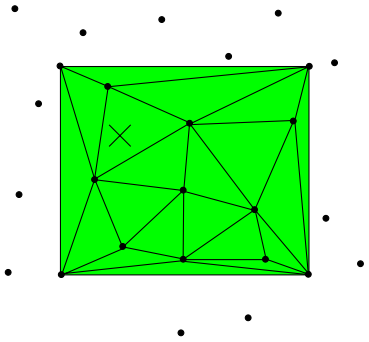


Figure 4.14: Estimating a value within the ROI.

So far the role of the hierarchy for the procedural generation of a simplical mesh has been not addressed in this section. As mentioned before, the first two approaches for generating a local mesh do not allow an easy straightforward incorporation of the hierarchy. Fortunately, with the last approach this can be easily achieved. The basic idea is to represent the vector field within the ROI with a fixed number of samples. This has the effect, that the effort necessary for the visualization is independent of the size of the ROI. Furthermore, this has the effect, that choosing a smaller ROI automatically increases the accuracy of the representation, since the same number of samples is used to approximate a smaller region.

Keeping the amount of samples in the ROI constant is achieved by using the clustering tree and the thread representing a hierarchy level explained in Section 4.1. In the beginning, the complete vector field is chosen as ROI and approximated by a configurable number of samples N_{usedSamp}. This is done by reproducing the cluster splits until the N_{usedSamp} are used to represent the vector field. Since initially the complete vector field is chosen as ROI, this is accomplished

by using all pre-computed splits up to split number $N_{\text{usedSamp}} - 1$. If the user chooses a ROI, a different refinement routine is used. Since not all split operations yield samples in the ROI, it is no longer possible to use the simples correspondence, $N_{\text{usedSamp}} = lastUsedSplit + 1$. Instead, it is now necessary to separately keep track of the number of samples currently within the ROI. If a pre-computed split is reproduced, the number of samples in the ROI changes depending on whether the representant of the original cluster was within the ROI and on whether both, only one or none of its descendants has a representant in the ROI. If the representant of node describing the current split process is not within the ROI, the number of samples in the ROI increased by the number of descendants (0, 1, or 2) that have a representant in the ROI. If the representant of the node is in the ROI, the number of samples in the ROI is only increased by one if both the representant of both descendants are in the ROI. In that case at least one of the descendants should be in the ROI so that it is not necessary to decrease the number of samples currently in the ROI. The split processes are reproduced until a number of $n_{usedSamp}$ is in the ROI, or all split operations were used, and the current hierarchy thread only consists of leaf nodes in the clustering tree.

We note that this approach refines the complete vector field until the desired number of samples is in the ROI, although only those samples are used afterwards. Section 6.3 describes a way that avoids refining the rest of the vector field and only generates a fine representation for the ROI. We further note that it is also possible to countermand the refinement of vector field, by successively replacing the two sons of the last split node by their father and correspondingly updating the number of samples in the ROI. Whenever the ROI is changed, the number of samples still in the ROI are counted. After that, depending if there are too much or too less samples in the region, the split steps are countermanded or further split steps are reproduced. The result is a ROI, that contains, if possible, a constant number of samples, regardless of its size or position.

4.4 Additional applications of the hierarchy generated by the clustering process

An additional application of the hierarchy generated by the clustering methods is its use for finding seed points for the placement of stream lines. Since the cluster representants are chosen in a way, such that each new representant is placed in the region, that deviates the most from the current vector field description, the resulting hierarchy can also be used to automatically generate seed points for stream lines. If the hierarchy is used in this way, then it is refined to a level containing the same number of samples as stream lines that should be placed in the vector field. The representants in that hierarchy level are used as seed points of stream lines. These stream lines can be computed using either the original vector field or a considerable higher level of the hierarchy.

4.5 Visualization on the Immersive WorkBench

The eventual goal of this work is to interactively and quickly visualize vector fields. This was done using the Responsive Workbench to create a virtual environment allowing to examine the vector field. The currently existing system is a prototype used to study how a visualization system can utilize the Responsive Workbench. For the implementation, the Immersive WorkBench from Fakespace, Mountain View, California, USA, a variant of the Responsive Workbench, was used. The implementation is based on Silicon Graphics's *OpenGL*, OpenInventor, and on the VLIB supplied by Fakespace with the Immersive WorkBench. Currently, widget sets, *e.g.*, Multigen SmartScene, are becoming available, allowing high-level user interface objects, and simplifying the task of developing an interface for an Immersive WorkBench based application.

The used VLIB only provides low-level constructs to control the Immersive WorkBench. It provides functions to open a graphics connection to the Immersive WorkBench, the tracker driver and the pinch driver. After doing this, the program should call the function `Bench_Draw()`. This function expects a pointer to a structure describing the current connection to the Immersive WorkBench, a pointer to a display function, and a pointer to a swap function. The display function is a user-defined function, which does the actual drawing. This function is called with a pointer to the already mentioned structure describing the Immersive WorkBench, and two integers. The first integer variable is the viewer number (currently either zero or one) and the eye number for the viewer (either zero or one). This display function is called for each eye of each viewer. Depending on the Immersive WorkBench configuration (stereo viewing, no stereo viewing, one or two viewers) this function is called one two or four times for each frame. The swap function is used to swap the frame buffers when double buffering[3] is used. If "`NULL`" is specified as a swap function, a standard function is used. Because it is unknown how often the user-defined display function is called, it is advisable not to use this function for manipulating data. Furthermore, this function is time critical, since it has to be called several times for each frame, and if one call takes too much time, this creates visible effects, which might disturb the user. For these reasons an approach using two threads was chosen. This is depicted in Figure 4.15. The two threads are shown as green boxes. The "tracker thread" reads the sensor data provided by the tracker and pinch glove drivers, evaluates it and manipulates the current object data base (blue circle) accordingly. The "display thread" uses the common data (or rather a part of it, namely lists of currently visible objects) and renders this continuously to the frame buffer. This is done in the main loop:

```
while ( Bench_Draw( brec , theDisplayFunction , NULL)!= 0 && !quit );
```

"`quit`" is a flag, which is set by the "tracker thread" when the user chooses "Quit" in the menu. The display thread is also responsible for terminating the application. This is done because the Fakespace library detects keystrokes, and `Bench_Draw()` returns a value different from zero when the user hits the "Escape" key. This terminates the main loop and the "display thread" sends a termination signal to the "tracker thread" using the Unix system call `kill()`.

The Fakespace VLIB also provides only elementary functions for getting input data from the

[3] A description of technical terms related to computer graphics, like *double buffering*, can be found in [44] and [6].

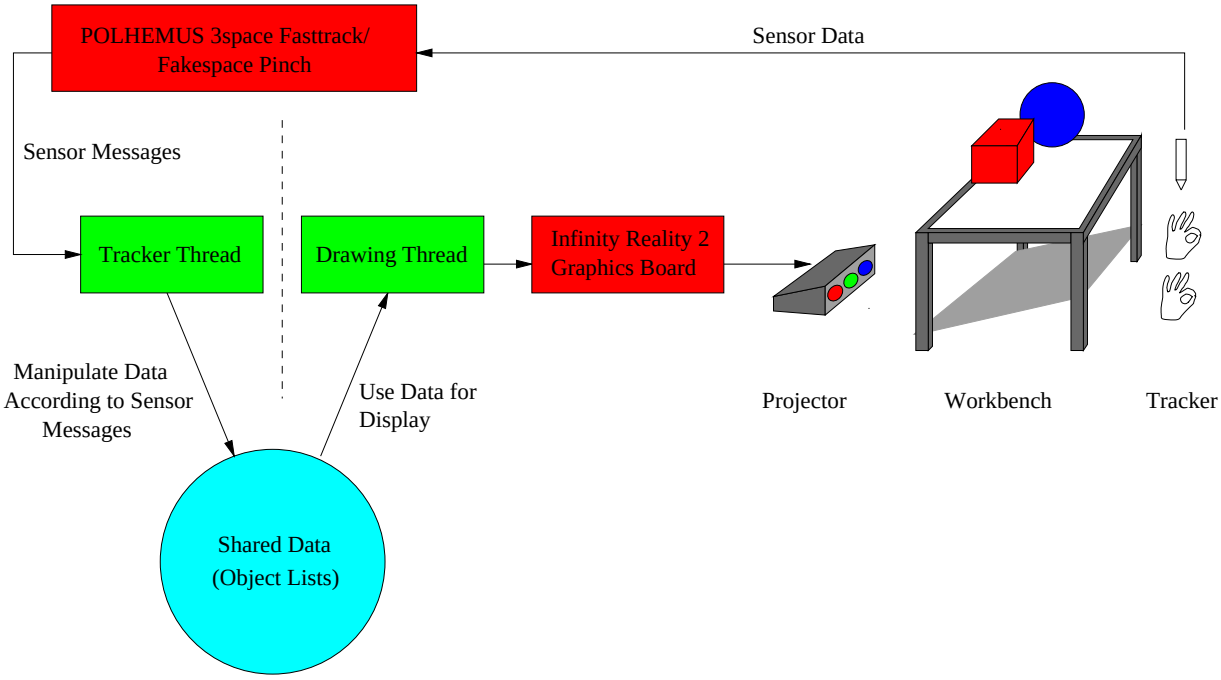


Figure 4.15: Two threads for using the Responsive Workbench.

trackers and pinch gloves. The function `GetTrackerMatrix(int deviceID, int whichReceiver, float tracker[4][4], void *data);` is used to get the position of the trackers. "`deviceID`" is an identification number that is obtained when the Immersive WorkBench and trackers are initialized. "`whichReceiver`" denotes the number of the tracker, whose position is to be queried. "`tracker`" is a pointer to a 4×4 array, which is filled with a matrix that converts from a local coordinate system that is "attached" to the tracker to the world coordinate system. This matrix can be used to obtain the position and orientation of the corresponding tracker. "`data`" is pointer to a variable that can be used to get additional information for this tracker. Currently this is only used for the stylus to report the status of its button as an integer flag (1 corresponds to a pressed button).

Pinch glove gestures can be determined with the function `QueryTouch(int whichDevice,int *gesture);`. "`whichDevice`" is an identification number that is obtained during the initialization of the Immersive WorkBench. "`gesture`" is a pointer to an integer variable that is used to return a gesture. This function returns an integer, that indicates whether the pinch gesture changed since the last call of this function and whether "`gesture`" contains the value of a new gesture.

These functions are used in the "tracker thread" to query the current status and modify the VE accordingly. This thread is described in Section 4.8. Since the "tracker thread" and the "display thread" work in the same address space using common data structures, it is necessary to synchronize their access to these data structures to avoid inconsistencies when the "tracker thread" modifies them. This is accomplished by using *semaphores*, a standard synchronization

method described, *e.g.*, in [29].

4.6 The “watch” menu

The watch menu provides a simple user interface in a virtual environment. It is based on an idea by Antonello Uva, a former Visiting Researcher at CIPIC and was implemented by Oliver Kreylos, a CIPIC PhD student. This thesis uses its own, enhanced implementation of this concept. The basic idea is to eliminate a static menu in the virtual environment and replace it by a menu that only appears when the user needs it. An intuitive method for accomplishing this goal is conjuring up the menu whenever the user looks on an imaginary watch. If the user wants to make a selection he just looks at the back of his hand and a menu appears. Using the stylus it is possible to make this selection.

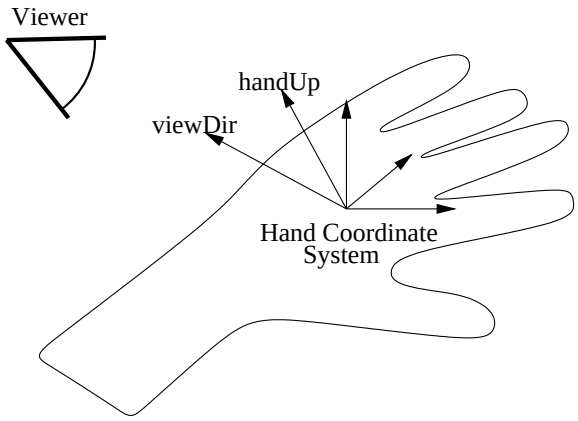


Figure 4.16: Determining whether the “watch” menu should be displayed.

Implementing this behavior is straightforward. The fundamentals are displayed in Figure 4.16. The vector “`handUp`”, defined in hand coordinates, points into the direction perpendicular to the back of the hand. “`viewDir`” is a vector that points from the origin of the hand coordinate system[4]. Both vectors are normalized, so that the scalar product of them can be used to compute the cosine of the angle between the vectors. If the scalar product is greater than 0.9 (corresponding to an angle of $25°$), the “watch menu” is displayed.

The implementation used in this work provides a class “`WatchMenu`”, that allow the creation of a menu. As soon as a “`WatchMenu`” object is created, it is possible to add buttons with the method `addButton()`. The buttons are arranged in rows with a user-specifiable amount of columns. For each button, a text can specified, that is displayed within. This is accomplished by using an OpenInventor “`SoText3`” node that is decomposed in triangles using the “`SoCallbackAction`”. These triangles are stored in an array and rendered within the button.

[4]The coordinate system defined by the tracker on the left hand.

In many cases it is desirable to have context sensitive menus. This prevents cluttering the main menu with options that only make sense if an object of a certain type is selected. In the current implementation this is accomplished by having a pure abstract base class "`MenuProvider`". Each object, that allows context sensitive selections, is derived from this class. So it is possible to use the C++ dynamic cast `dynamic_cast<MenuProvider>` to determine if the currently selected object provides an own method. This class is defined in `MenuProvider.h` and shown in Listing 4.1.

Listing 4.1 The Class MenuProvider

```
// Abstract class for an object, that provides a menu
class MenuProvider {
  WatchMenu *objMenu; // Cache for menu
  public:
  MenuProvider() : objMenu(NULL) {}
  virtual ~MenuProvider() { delete objMenu; }
  // Create a menu
  virtual WatchMenu *createContextMenu(void)=0;
  virtual WatchMenu *getContextMenu(void);
  // Handle the selection
  virtual void handleMenuSelection(int sel)=0;
};

inline WatchMenu *MenuProvider::getContextMenu()
{
  if (objMenu)
    return objMenu;
  else
    return (objMenu=createContextMenu());
}
```

The pure virtual method `createContextMenu()` has to be overwritten by the derived object in order to create the menu corresponding to that object. If the user selects one of the menu buttons, the method `handleMenuSelection()` is called. This method must be defined in the derived class as well to achieve the desired behavior. The method `getContextMenu()` is used by the "tracker thread" to generate a menu for the current selection. The default behavior is to create this menu at the first invocation of `getContextMenu()` and to cache this menu in "`objMenu`". Further invocations of `getContextMenu()` simply return the already created menu. If the menu should reflect the current state of the object, this method has to be overwritten by the derived class.

In the current implementation, the global variable "`currentMenu`" contains a pointer to the menu belonging to the currently selected object or the root menu, whenever no object is selected. "`watchMenuVisible`" is a global flag that is set whenever the user looks at his hand. These two variables are used by the "display thread" to display the menu when necessary.

4.7 World space and vector field space

In order to simplify the display of vector field data, the current implementation uses three coordinate systems:

1. world coordinates,
2. model coordinates, and
3. vector field coordinates.

"World coordinates" refer to the coordinate system used by VLIB and describe the Immersive WorkBench table top and the space around it. One unit in this coordinate corresponds to one inch (2.54 cm) in the real world, *i.e.*, when displayed on the Immersive WorkBench. It is mainly used to display the watch menu discussed in the previous section. "Model coordinates" are used to implement the model-on-a-stick behavior described in Section 2.15. Its main use is to display manipulators, *i.e.*, handles used to manipulate the visualization objects in the vector field. "Vector field coordinates" simplify the computation of the visualization object, like stream lines. It frees the implementation of the visualization objects from the need to convert all coordinates in model-on-a-stick coordinates. This is also convenient, if a different region of the vector field is displayed. These changes are transparent to the visualization objects, and the implementation does not have to consider which current transformations are necessary to render in model coordinates. The vector field coordinates are given with respect to the model coordinates, so that modifying the orientation of the model also affects the display of the vector field. The manipulators are rendered in model coordinates to ensure that the current vector field transformation does not affect their size. The drawback is that the current transformation between vector field and model coordinates has to be used, when a manipulator is used to change, *e.g.*, the position of a visualization object.

4.8 The "tracker thread"

The "tracker thread" is responsible for the interaction between user and VE. All sensor data are continuously queried, and the scene is updated accordingly. The tracker thread is spawned using the SGI IRIX system call `sproc()`, which is similar to the standard Unix call `fork()`, but allows both processes to share a single address space. This simplifies communication with the "display thread" since common data structures in memory can be accessed. The "tracker thread" consists of a main loop, that can be found in the function `trackerThread()`. This function consists of a continuous loop, that queries the sensor data and acts accordingly. The only way to abort this loop is the reception of a termination signal ("`SIGTERM`") sent by the display thread.

After querying the sensor data, the function `trackerThread()` checks, whether the watch menu is visible. If this is the case, then it checks whether an item in this menu is selected and highlights the corresponding menu button (by calling the corresponding methods of the class "`watchMenu`"). When the user just released the stylus button in a menu button, the

function `handleMenuSelection()` is called, which ensures to relay this fact to the correct function. This is either `handleEmptySelMenuSelection()`, if no object is select, or the method `handleMenuSelection()` of the currently selected object.

In the following step, the stylus data is translated into events. The following events are possible:

Stylus move The stylus is moved while its button is not pressed.

Stylus button push The button of the stylus has just been pushed.

Stylus drag The stylus is moved while its button is pressed.

Stylus button release The stylus button has just been released.

These events are handled by calling the corresponding functions, defined in `HandleEvents`. *E.g.*, for a stylus drag event the function `handleStylusDrag()` is called. In this module the actual modifications to the VE are performed. Each event handler function is called with the current position of the stylus as an argument. If the user performed a menu selection, or if the current stylus drag started in the menu (this is detected by setting a flag, when the stylus button is pushed while being inside a menu button), these events are ignored and not forwarded to the module `HandleEvents.C`.

The pinch glove events are handled directly by the function `trackerThread()`, since currently they are only used to change the model-on-a-stick transformation. These transformations are performed as described in Section 2.15. Additionally, it is possible to scale the model. If the user performs a pinch gesture with both thumbs and index fingers, the size of the model is increased or decreased proportionally to the distance between her/his hands. This is achieved by storing the distance between the user's hands and the current transformation from model to world coordinates at the beginning of the pinch gesture. When the user moves her/his hands, while keeping the pinch gesture, the distance between her/his hands, and a corresponding scale matrix are calculated and the current model transformation is replaced with the product of this scale matrix and the model matrix stored at the beginning of the pinch gesture.

The module `HandleEvents.C` is responsible for the actual modifications to the virtual environment. This environment consists of objects. Each object is derived from the class "`StylusObject`", which is defined according to Listing 4.2

The implementation of `HandleEvents.C` is based on the concept of a state machine. One possible state is "`idle`". In this state all events are relayed to the currently existing objects. Each "`StylusObject`" has three different states of its own ("`inactive`", "`highlighted`", and "`active`"). Normally all objects are "`inactive`". In this state, all stylus moves are observed. If the stylus is within an object (tested with the method `within()`), it is highlighted (by calling the method `highlight()`) to inform the user that pressing the stylus button at that location will activate the object. If the stylus leaves the object, *i.e.*, `within()` no longer yields true, the highlighting is deactivated by calling the method `unhighlight()`. If the user pushes the stylus button, then the object receives an `activate()` message. Furthermore its method `beginDrag()` will be called. During the drag process, the current object's

Listing 4.2 The abstract base class "`StylusObject`"

```
// Abstract class for an object that can be manipulated by the
// stylus
class StylusObject {
  public:
    typedef enum { inactive, highlighted, active } status;
  protected:
    status objectStatus;
  public:
    StylusObject() : objectStatus(inactive) {}
    virtual ~StylusObject() {}

    // Check, if position is within the object. Meaning, that if
    // the button is pressed, the object should be selected
    // and activated
    virtual bool within(const Vertex &p)=0;

    // Change object state
    virtual void activate(const Vertex &) { objectStatus=active; }
    virtual void deactivate(void) { objectStatus=inactive; }
    virtual void highlight(void) { objectStatus=highlighted; }
    virtual void unhighlight(void) { objectStatus=inactive; }

    // Handle dragging
    virtual void beginDrag(const Vertex &) {}
    virtual void handleDrag(const Vertex &) {}
    virtual void endDrag(const Vertex &) {}
};
```

`handleDrag()` method is continuously called. When the button is released, `endDrag()` is called. `deactivate()` is called, when the user presses the button outside the active object.

If `HandleEvents.C` is in the state "`deleteObject`", the highlighting continues as mentioned in the previous paragraph. But if the stylus button is pressed within an object, that object is deleted rather than activated.

New objects can be created when `HandleEvents.C` is in the state "`createObject`". To simplify the addition of new visualization objects, the events during the object creations are delegated to a template for the new object. Object templates have to be derived from the class "`ObjectTemplate`" shown in Listing 4.3. Whenever a new object is to be created, a corresponding template object is created, as well as a pointer to this template stored in "`currentTemplate`", and the state set to "`createObject`". This causes all stylus events to be passed to this template. At each release of the stylus button this template has the possibility to end the object creation process by returning a pointer to a newly created "`StylusObject`". This is added to the list of current objects, the template is deleted, and the state is set back to "`idle`".

Listing 4.3 The abstract base class "`ObjectTemplate`"

```
class ObjectTemplate {
  public:
    virtual void handleStylusMove(const Vertex &p) {}
    virtual void handleStylusButtonPush(const Vertex &p) {}
    virtual void handleStylusDrag(const Vertex &p) {}
    virtual StylusObject *handleStylusButtonRelease(const Vertex &p)=0;
    virtual ~ObjectTemplate() {}
};
```

Selecting a region, *e.g.*, a ROI, is done in the state "`selRegion`". In this case the events are handled by `HandleEvents.C` itself and not relayed to any object. In this state a transparent box, denoting the current region, is drawn.

Implemented subclasses of "`StylusObject`" are "`moveManipulator`", "`StreamLine`" and "`StreamRibbon`". "`moveManipulator`" is a simple object, displayed as a box, that can be moved with the stylus. It is used as a building block in "`StreamLine`" and "`StreamRibbon`" to allow easy manipulation of these visualization objects.

4.9 The "display thread"

The "display thread" is responsible for displaying the VE on the Immersive WorkBench. In order to do this, it continuously displays the current objects. These objects are organized in two lists, each corresponding to a coordinate system described in Section 4.7. In order to increase the efficiency of the rendering process, the rendered objects are cached in OpenGL display lists. The corresponding classes are shown in Listing 4.4. "`RenderObject`" is the base class for all objects that are displayed. It provides a simple caching scheme by using an OpenGL display list and a flag denoting whether the current cache content is valid. Derived from this class are the classes "`VectorfieldRenderObject`" and "`WorldRenderObject`". This distinction, along with the different names for the methods that perform the actual rendering (`glRenderInWorld()` and `glRenderInVectorField()`), allow visualization objects to be derived from both classes and display different graphics primitives within those two coordinate systems.

The caching mechanism is shown for the class "`VectorFieldRenderObject`" in Listing 4.5. The implementation for "`WorldRenderObject`" is analogous. When an object is rendered for the first time it is simultaneously written to a OpenGL display list. For further rendering of the object this render list is accessed instead of the object. If the object changed, *e.g.*, moved to another position, since it was written in the display list, the method `invalidateCache()` must be called to avoid that an old representation of the object is displayed.

The class "`InventorObject`" allows displaying OpenInventor objects along with the OpenGL primitives. This is used for displaying an Inventor object, that represents the geometry belonging to the vector field, *e.g.*, the object for which the flow field is visualized. This

Listing 4.4 Base classes for rendered objects.

```
class RenderObject {
  protected:
  mutable GLDisplayList renderCache;
  mutable bool cacheValid;

  public:
  RenderObject() : renderCache(), cacheValid(false) {}
  void invalidateCache() const { cacheValid=false; }
};

// Object to be rendered in world coordinates
class WorldRenderObject : public RenderObject {
  public:
  virtual void glRenderInWorld(void) const=0;
  void renderGL(void) const;
};

// Object to be rendered in vector field coordinates
class VectorFieldRenderObject : public RenderObject {
  public:
  virtual void glRenderInVectorfield() const=0;
  void renderGL(void) const;
};
```

OpenInventor geometry is consequently displayed in the vector field coordinate system.

If the flag "`menuVisible`" is set by the "trackerThread", the watch menu which is referenced by the pointer "`currentMenu`", is displayed. The currently selected region (if there is one) and the watch menu are transparent, so special care has to be taken when drawing them. First all solid objects are drawn. Afterwards the *depth buffer*[5] is set to read-only mode. Subsequent drawing operation still alter the frame buffer, provided the specified z-coordinate is smaller than the corresponding depth buffer value. Since the depth buffer value is not changed, one transparent object does not occlude another transparent object, *i.e.*, prevents another object to be drawn, because its z-coordinate is bigger than the z-coordinate of the current object. Thus the colors of all transparent objects that are not occluded by opaque objects are blended together. To ensure satisfactory results, the primitives should be drawn from back to front, so that the blending yields the correct result. In the current implementation this can be achieved by first drawing the box denoting the ROI and afterwards the watch menu. Further details can be found in the Section "Three-Dimensional Blending with the Depth-Buffer" (see pp. 222–226 in [44]).

In one instance, a second rendering context is created by the "tracker thread" and used to render the current scene (without the ROI and the watch menu) to a buffer. This is used to

[5] See [44] and [6].

Listing 4.5 Caching rendered objects.

```
inline void VectorFieldRenderObject::renderGL(void) const
{
  if (cacheValid) {
    renderCache.call();
  }
  else {
    renderCache.newList(GL_COMPILE_AND_EXECUTE);
    glRenderInVectorfield();
    renderCache.end();
    cacheValid=true;
  }
}
```

enable the user to save the current visualization results to a file. If he chooses “Save view” in the watch menu, the current observer position is determined, a rendering context created, a camera positioned according to the viewers position and the visualization objects rendered to a buffer. This buffer is saved to a file. This enables the user to use the results from the visualization, *e.g.*, for illustrations in a paper or book.

Chapter 5

Results

5.1 Clustering results

The clustering algorithm was evaluated using several 2D vector fields. One of them was the field defined by the analytical function

$$F(x,y) = \left(-x^2 + xy + 0.5 * y + 0.25, x^2 + xy - 0.5 * y - 0.25\right) \quad . \tag{5.1}$$

This vector field has three critical points, *i.e.*, points with $F(0,0) = 0$ in the region $[-1,1] \times [-1,1]$. It is displayed in Figure 5.1. The data set contains 6000 samples resulting from evaluating equation 5.1 at random locations. The positions of the samples are denoted by the small filled circles. The direction and length of the vector at a given position is given by the line segment originating in the filled circle.

Figure 5.3 is the result of computing the Delaunay triangulation of the positions and applying linear interpolation over each triangle. Figure 5.3(a) shows the color coded vector length (blue indicating a length of zero and red denoting the maximum length vectors in the data set, the mapping between color and vector length is illustrated in Figure 5.2(a)). Figure 5.3(b) shows the same data, this time with the direction of the vectors coded as color (the corresponding mapping between colors and directions is shown in Figure 5.2(b), *e.g.*, a vector pointing to the top corresponds to red).

These discrete data sets are used as input for the clustering algorithm. First, the refinement of the second approach alternating between the distance measures is examined. For the creation of the examples, a split distance function weighting the angle with 2 and the length difference with 1 has been used, to prioritize the preservation of vector direction. Figure 5.4 shows the clusters after four split operations. The positions of the samples are denoted by small spheres, which are colored according to cluster assignment, *i.e.*, samples of same color belong to the same cluster. The shape of the clusters is similar to a Voronoi tessellation. A difference can be seen in the lower part of Figure 5.4. The boundary between the two lower most clusters is not a line as one would expect from a Voronoi tessellation. The reason for this is the fact that the samples are assigned to the clusters first and after that the center is recomputed as average of all positions in the cluster. The approximation resulting from using the representants from 400 clusters is shown in Figure 5.5. It shows the color-coded length (Figure 5.5(a)) and direction

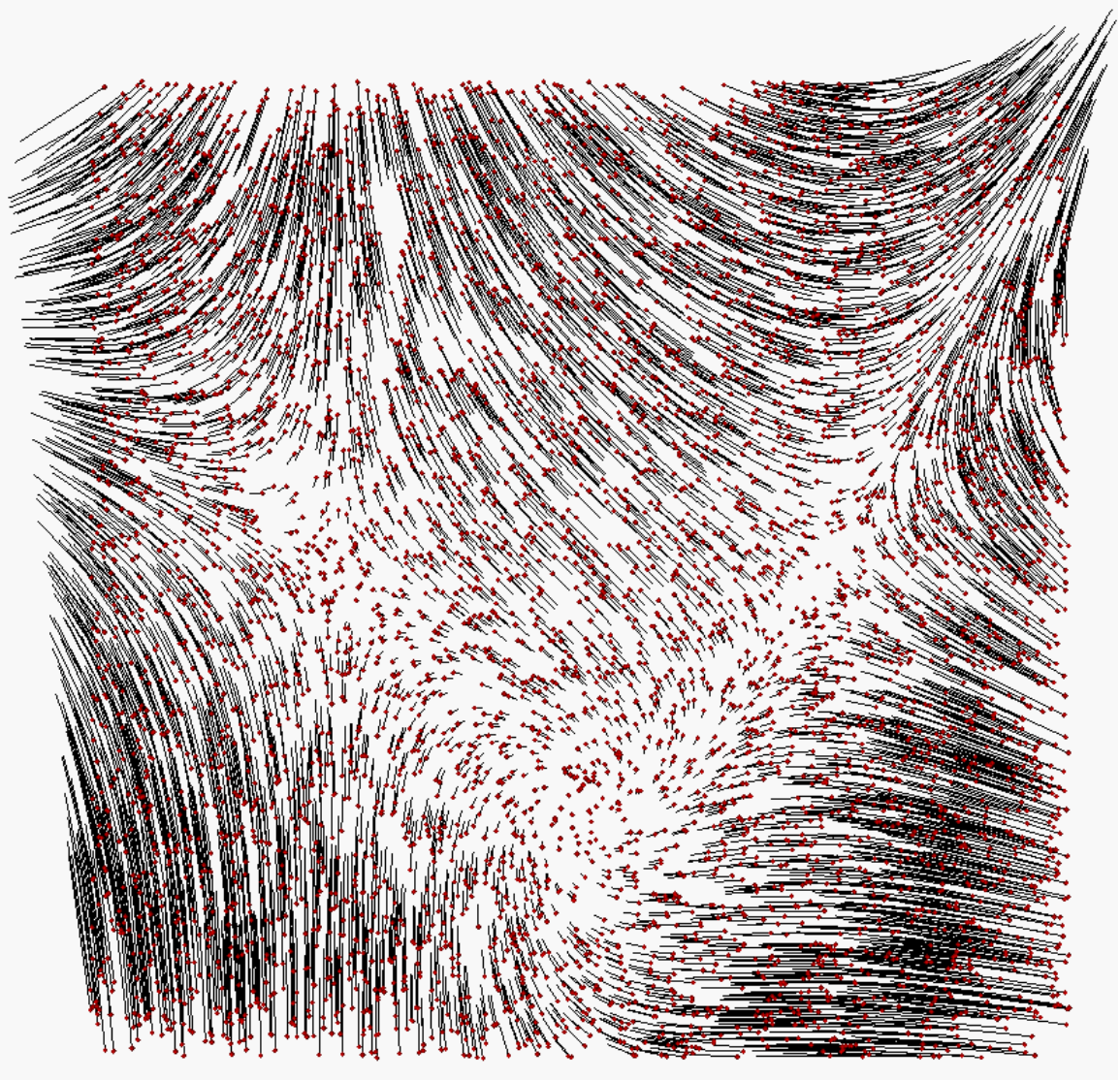
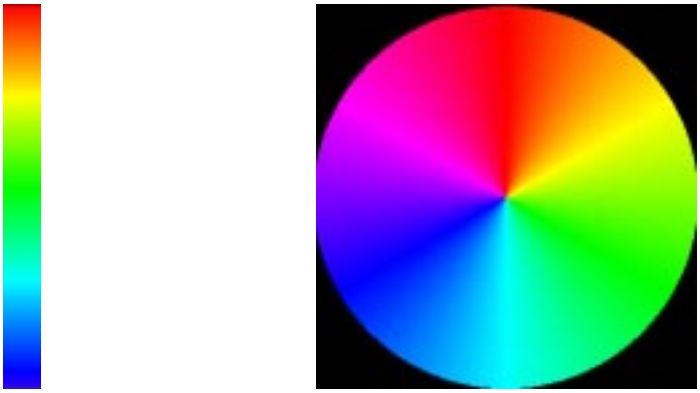


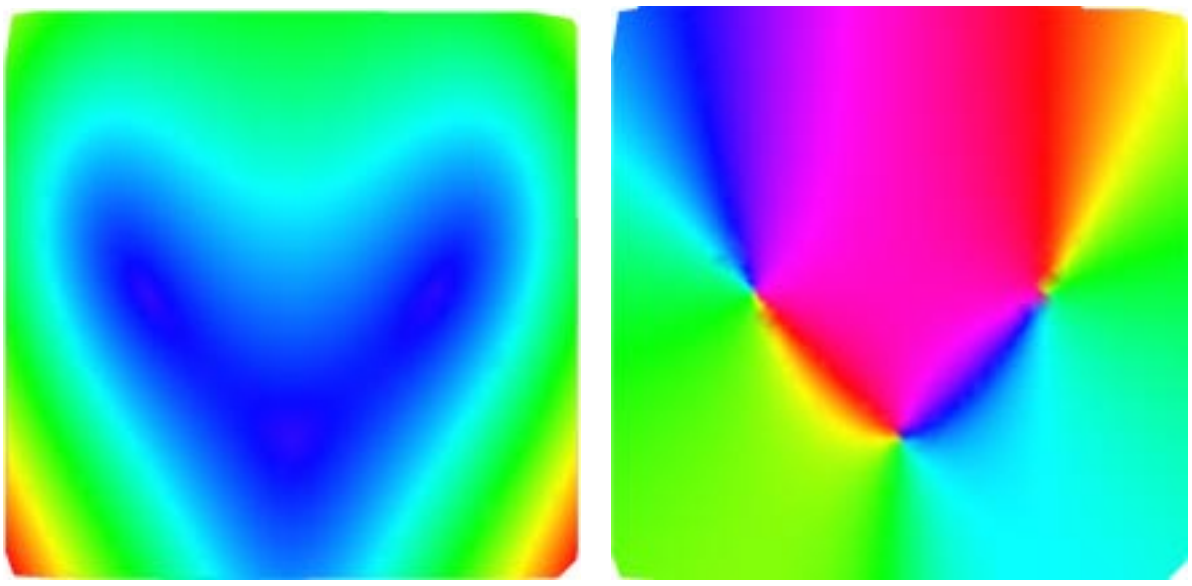
Figure 5.1: Discrete representation of the example vector field with three critical points.



(a) vector length — blue denotes a length of zero, and red denotes the maximum vector length

(b) vector direction — coded using “color wheel”

Figure 5.2: Color coding vector length and direction.



(a) color denotes vector length

(b) color denotes vector direction

Figure 5.3: Triangulated and linearly interpolated representation of the example vector field.

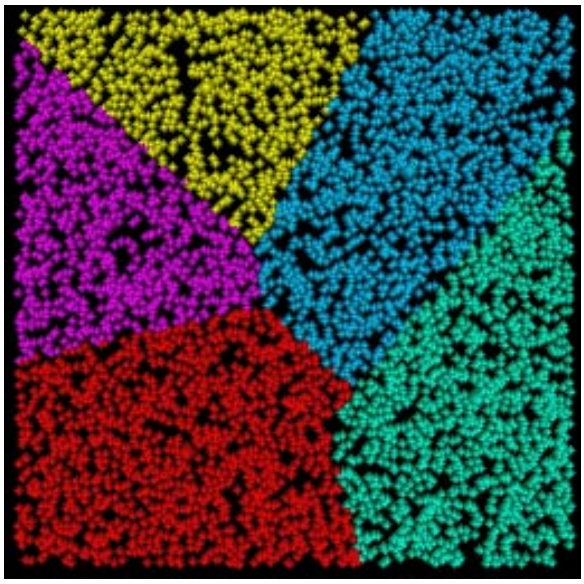


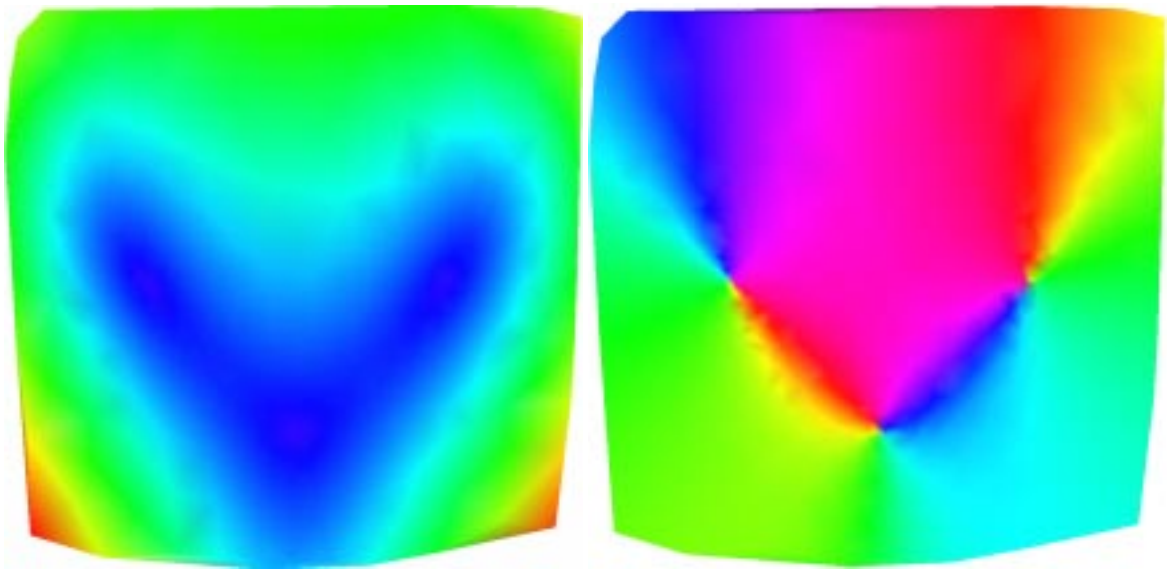
Figure 5.4: Clusters resulting from using approach II — refined by alternating use of distance functions.

(Figure 5.5(b)) of the approximation of the vector field by computing a Delaunay triangulation of the cluster centers and using linear interpolation. Although less than one tenth of the amount of original samples is used, a reasonable approximation can be achieved.

Figure 5.6 shows the underlying triangulation. We note, that the triangulation is finer in the vicinity of the critical points, where the variation of the vectors is higher. In regions of less variation a coarser representation is used. This is exactly the desired result.

Figure 5.7 shows the results using the second variation of clustering approach II. Here the assignment distance function was modified to be also dependent on the vector part. The figure shows the case of a poor choice of the weights (position weight $\lambda_1 = 2$, angle weight $\lambda_2 = 2$, length difference weight $\lambda_3 = 1$). In this case, the vector part dominates the distance function despite the fact that according to the discussion the position should be the dominating part. It nicely illustrates that this causes incoherent clusters. Vectors from different regions of the vector field are assigned to one cluster. This remains the case even when more clusters are generated. This is illustrated in Figure 5.8, which shows the approximation of the vector direction, again using the representants of 400 clusters. Near the critical points one can see several artifacts (spots of different color) that are caused by clusters containing samples from different sides of the critical points.

These problems disappear when the position clearly dominates the assignment distance measure. This is illustrated in Figure 5.9. We note that all clusters are coherent now. This is because the Euclidean distance between the positions of samples now clearly dominates the distance measure. However, the clusters are still shaped in a way to resemble regions of vectors of similar direction and length. The approximation resulting from using the representants of 400 clusters of this approach are illustrated in Figures 5.10(a) and 5.10(b). Again, a reason-



(a) color denotes vector length (b) color denotes vector direction

Figure 5.5: Approximation of the vector field with 400 clusters using the first variation of approach II.

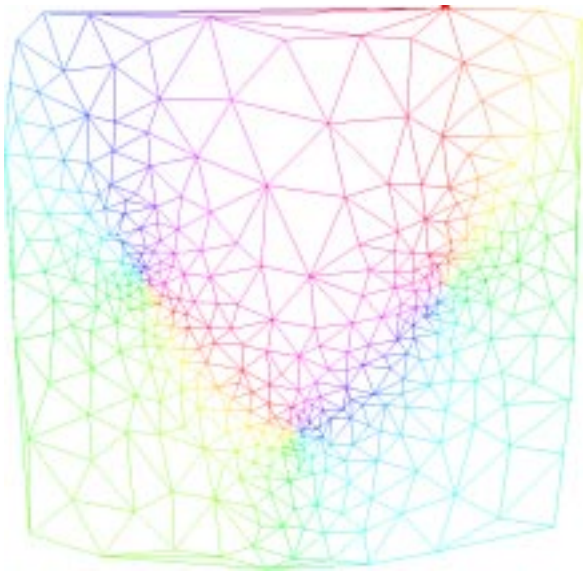


Figure 5.6: Triangulation of the approximation of vector direction using the first variation of approach II.

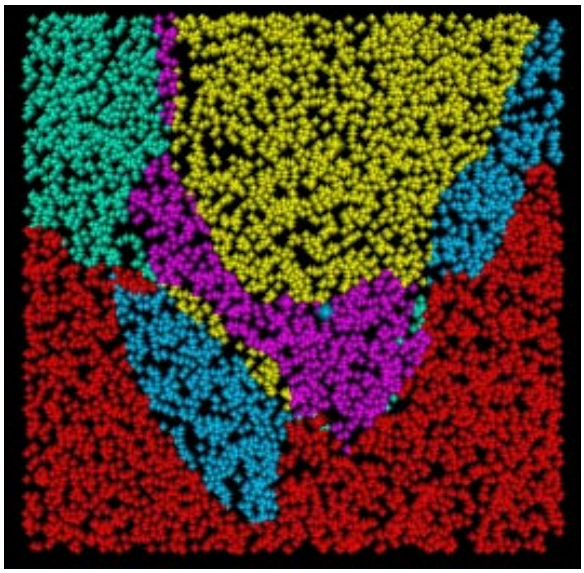


Figure 5.7: Clusters resulting from using II — refined by adding vector dependency to the assignment distance with poorly chosen weights.

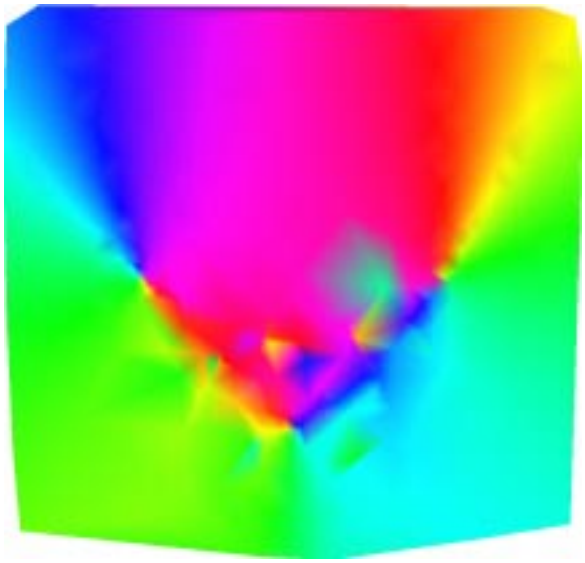


Figure 5.8: Artifacts in vector field approximation due to poorly chosen weights.

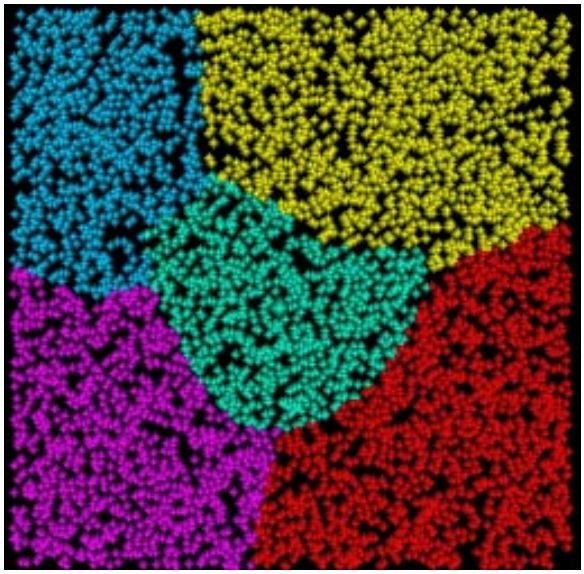
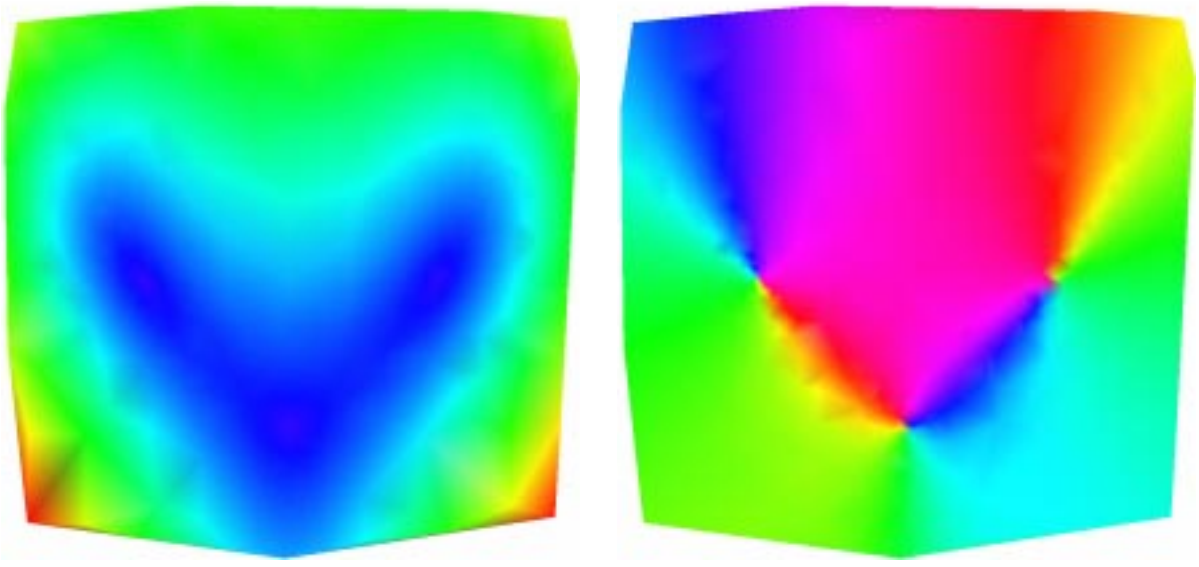


Figure 5.9: Clusters resulting from using approach II — refined by adding vector dependency to the assignment distance with improved weights.



(a) color denotes vector length

(b) color denotes vector direction

Figure 5.10: Approximation of the vector field using the second variation of approach II.

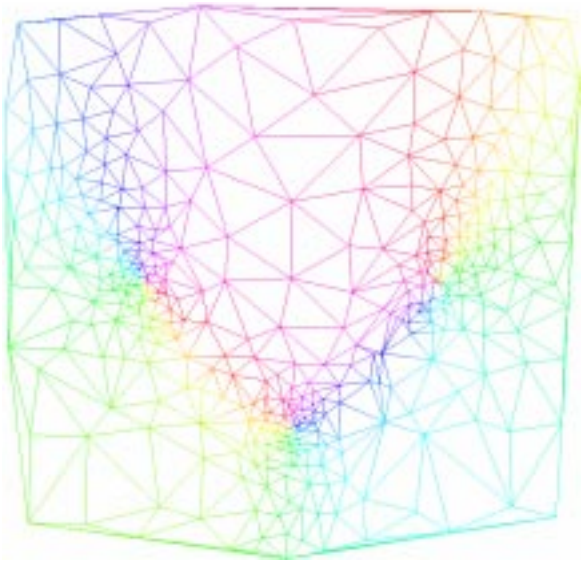


Figure 5.11: Triangulation of the approximation of vector direction using the second variation of approach II.

able approximation is achieved. At close examination some minor artifacts are visible. This is due to the fact that the interpolation was not modified to fit this approach as suggested in Section 6.2. The underlying triangulation is shown in Figure 5.11. Regions of high variation are approximated by relatively more triangles.

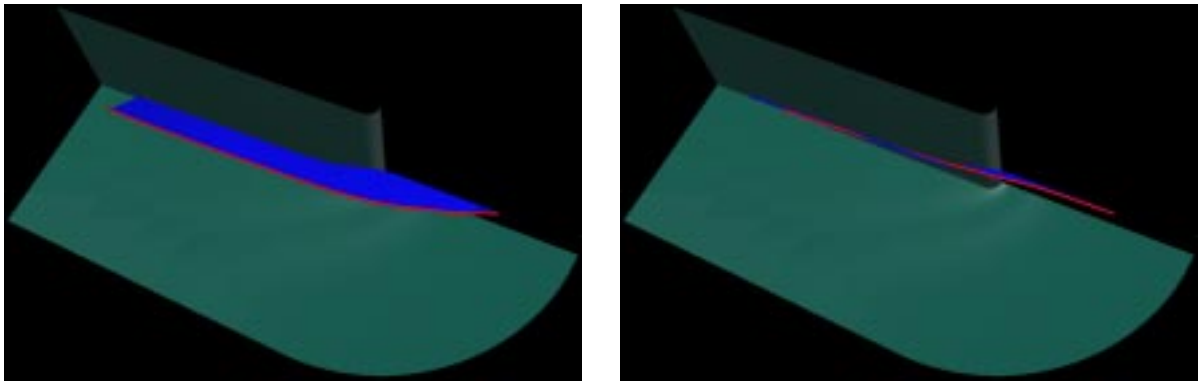
5.2 Visualization results

The results of the algorithms for utilizing the hierarchy for visualization purposes presented in Chapter 4 are discussed in this section. For the purpose of demonstrating these algorithms, data sets resulting from the improved clustering algorithm discussed by Heckel *et al.* [16] are used. Figure 5.12 shows stream lines in the "blunt fin" data set[1] using an increasing number of representants. The original data set consists of 40921 samples. The stream line computed using the original data set is displayed in blue, the steam line computed using the approximation is drawn in red. The difference surface between these stream lines is shown in blue. Even with merely 5001 samples a reasonable approximation is achieved. The complete hierarchy can be accessed using a slider that seamlessly blends between the different hierarchy levels.

Figures 5.13 shows approximations of stream surfaces at increasing resolutions. Again, the stream surface computed using the original data set is displayed in blue, the one computed using an approximation is displayed in red.

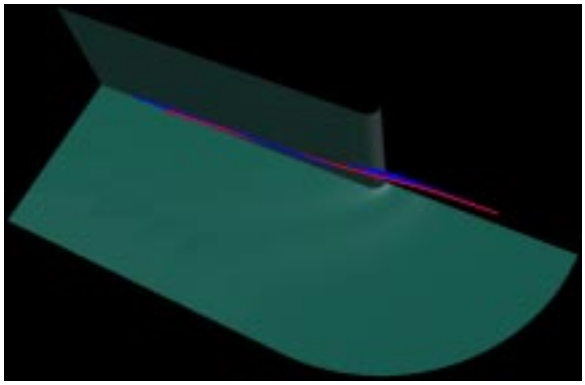
The hierarchy generated in the clustering step can also be used to generate "seed points" for stream lines, *i.e.*, points at which the stream lines begin. This is illustrated in Figure 5.14.

[1] Compare Appendix B.3.



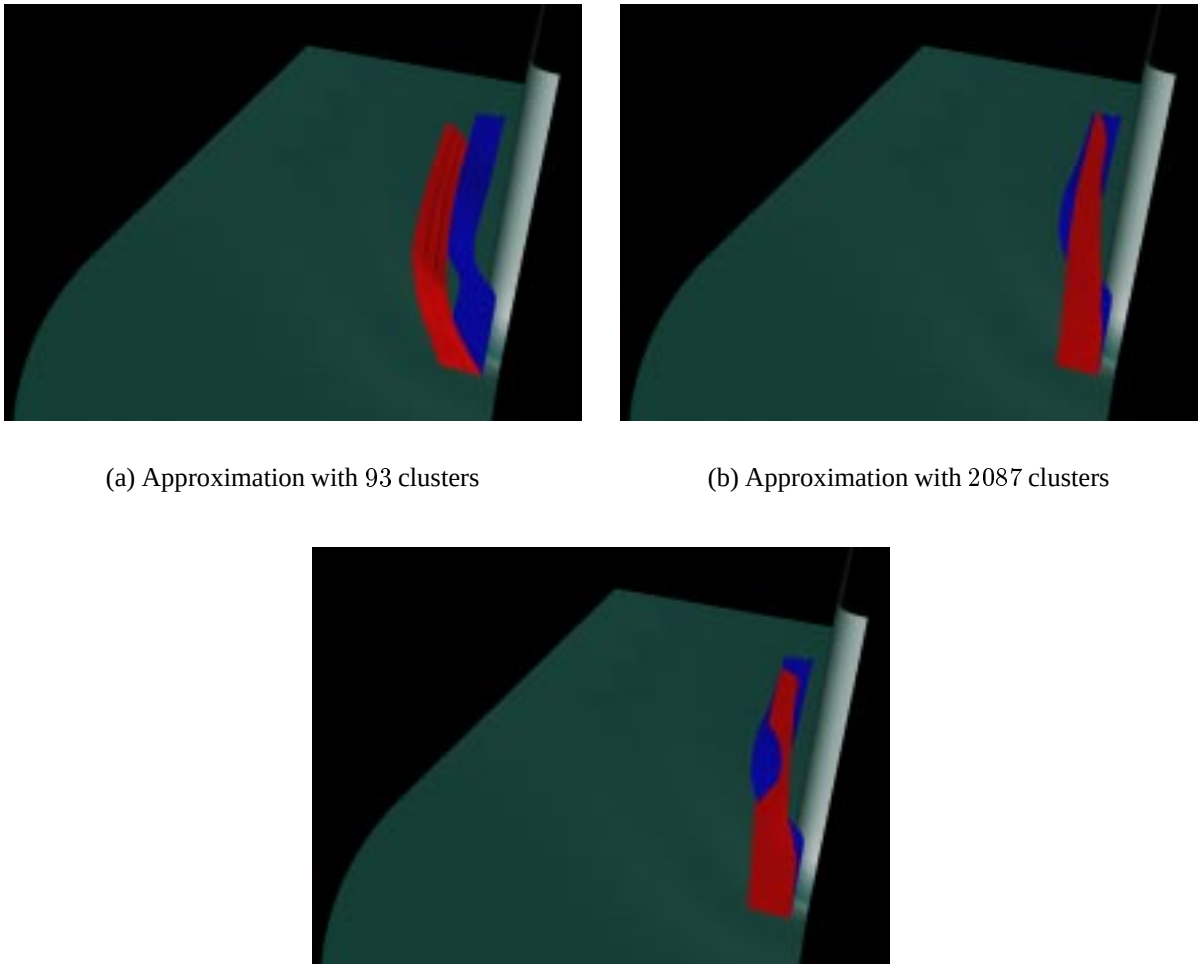
(a) Approximation with 62 clusters

(b) Approximation with 2087 clusters



(c) Approximation with 5001 clusters

Figure 5.12: Stream lines for “blunt fin”-data set — blue stream lines are computed using the original data set; red stream lines are computed using an approximation.



(a) Approximation with 93 clusters

(b) Approximation with 2087 clusters

(c) Approximation with 5001 clusters

Figure 5.13: Stream surfaces for “blunt fin”-data set — blue stream surfaces are computed using the original data set, red stream surfaces are computed using an approximation.

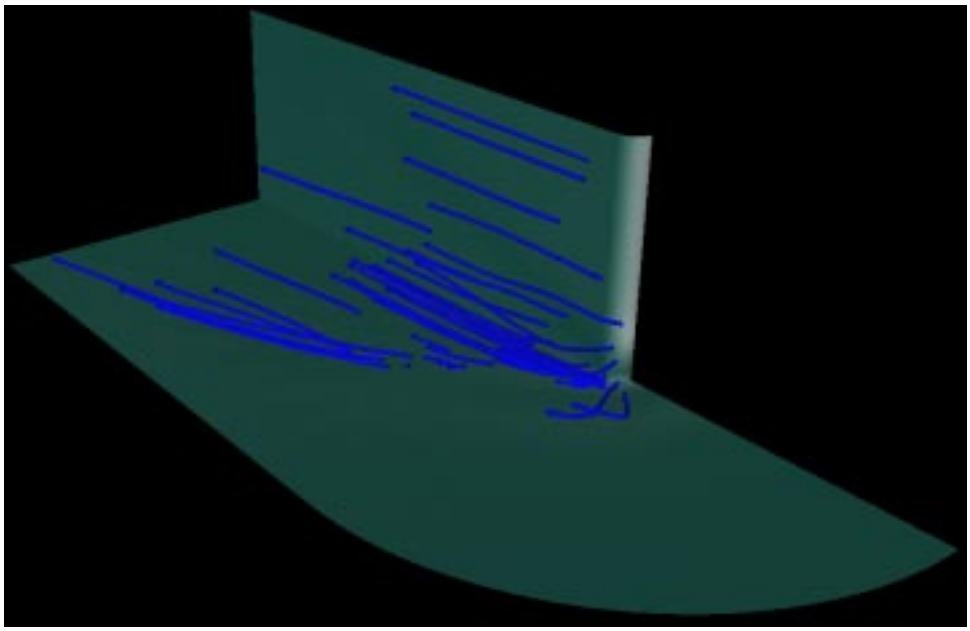


Figure 5.14: Utilizing the hierarchy to find “seed points” for stream lines.

This Figure shows 50 stream lines, computed using the original data sets. The start points of the stream lines are the representants of the hierarchy. Despite of the relatively small number of stream lines, interesting features of the flow, *e.g.*, its reversed behavior near the plate (the two rightmost stream lines in Figure 5.14), are detected.

Figure 5.15 shows the local triangulaiton produced by the first approach for the procedural triangulation discussed in Section 4.3. Only a localized triangular mesh consisiting of the 20 closest points to the query point is generated. This number is sufficiently small to move the resulting mesh interactively on SGI Origin with an Infinity Reality 2 graphics board using one of its four processors. However, since each point query results in the computation of a new mesh, moving the stream line is too slow. One stream line, computed using the first procedural mesh geneartion approach is drawn as a red line. We note that in this case line segments instead of tubes are drawn. This illustrates that rendering stream lines with cylindrical primitives really yields an improvement of display quality.

Figures 5.16 and 5.17 show triangulations using the ROI approach. The ROI in Figure 5.17 is considerably smaller than the ROI in Figure 5.16. It is clearly visible, that this results in a finer triangulation within that region. Both triangulations contain 100 samples. This allows a quick recomputation of the triangulation and an interactive redefinition of the ROI.

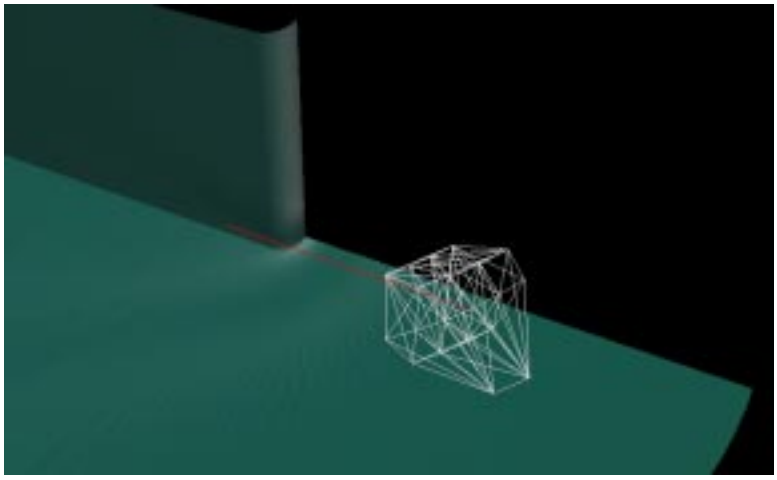


Figure 5.15: Local triangulation of the samples.

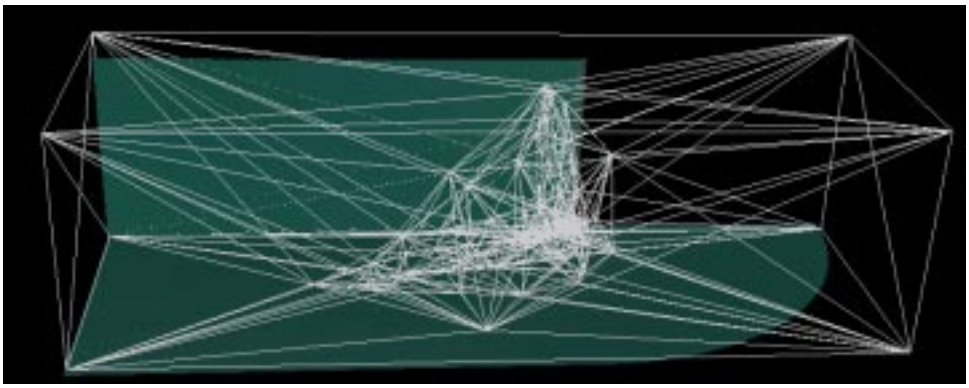


Figure 5.16: Triangulation within a relatively large ROI.

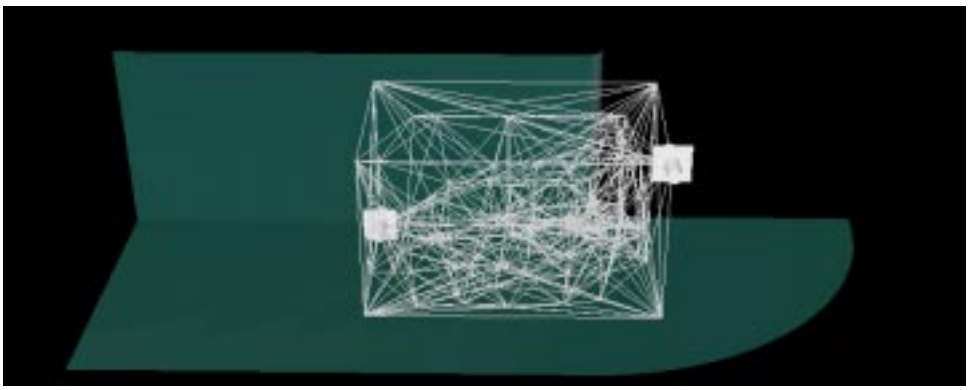


Figure 5.17: Triangulation within a relatively small ROI.

Chapter 6

Future work

6.1 Outlook

As mentioned in the Preface and Introduction sections the field of large-scale data set visualization is still in its infancy. The same is true for the field of visualizing large-scale vector field data sets. This work presents some first approaches to solve the underlying problems of this important new field. However, like in any pioneering field, there is plenty of work left.

6.2 Improving the clustering process

The clustering process was already improved by Heckel *et al.* [16] and the results have been partially used in the visualization part of this Diplomarbeit. Nonetheless, there is still need for improvements in the clustering process. Furthermore, the fundamental approaches discussed in this Diplomarbeit can be enhanced, yielding alternatives to the algorithm described in [16].

If clustering is used in conjunction with grids, then it should be possible to take the connectivity of the grid into account. A first approach could modify the neighborhood relation to use the grid and prevent samples to be assigned to a cluster, when their grid cells are not connected. This, however, has major implications and the top-down approach that initially assigns all points to one cluster.

An even more fundamental point is the preservation of boundaries of a vector field. This poses no problem when Hardy interpolation is used. Hardy's method allows querying points at arbitrary positions, particularly positions outside the convex hull of the sample points forming the scattered data set. If a tetrahedral mesh is used this is not the case anymore. A tetrahedral mesh only defines interpolation value within the convex hull of the positions used as its vertices. This is due to the fact (mentioned in Section 2.5) that a triangulation is a decomposition of the convex hull into simplices. Thus its boundary coincides with the convex hull of the data set. To approximate a value at a specified position, the simplex containing it is located. Thus it is only possible to approximate values within the boundary of the triangulation. The problem arising from this fact is illustrated in Figure 6.1. It shows a simple vector field and its cluster representants. It is obvious that the convex hull of the cluster representants (colored red) is completely

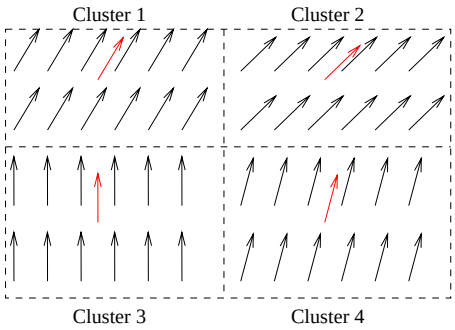


Figure 6.1: Different convex hulls of original data set and cluster representants.

within the convex hull of the samples. Any cluster representation yields a convex hull that is smaller than the original convex hull. This is due to the fact that the cluster representants are placed at the centers of clusters. If a cluster at the border of the vector field, *i.e.*, a cluster that is adjacent to the convex hull of the vector field contains more than one vector (which is almost always true), its center will lie within the original convex hull and thus the convex hull of all cluster representants will lie completely within the convex hull of the vector field. The resulting error in the boundaries of the simplified vector field is dependent on how much the vector field varies near to its boundary. If the changes are minimal, then the resulting clusters have a greater extent and their centers will be further from the original boundary.

In most cases, it is desirable to keep the boundary of the vector field or to have control over the approximation of the convex hull. In order to preserve it, the convex hull must be known. It is not feasible to compute the convex hull of the complete original data set, because its size is too big for this to be done in a reasonable amount of time. One possibility is to use an approximation algorithm for computing the convex hull, but this might still be too inefficient. A more promising approach can be derived from the fact that the problem of preserving the convex hull only arises in border clusters. So it is possible to determine the convex hulls of the boundary clusters (or an approximation to it when the border structure of the vector field should also be simplified). The points on the convex hull can be added to the triangulation in addition to the cluster representants to get an approximation of the entire region in which the vector field is defined.

In the case of a given grid, it might be possible to use a modification of the clustering approach described in Section 3.3. If connectivity information is available, it becomes possible to replace the second, position-based pass with a grid-based path. The first pass is performed as described in Section 3.3 and clusters only based on the vectorial part. Instead of performing a second, position based pass afterwards, it may be possible to use the connectivity information to partition the clusters resulting from the first pass in spatially coherent clusters. This could be done by starting with a seed sample and gradually adding all adjacent samples that are in the same cluster. If no more samples can be added that way, and the original cluster is still not empty, a different seed sample is chosen. The whole process is repeated until the original cluster is empty. When this is done for all clusters the result is a set of spatially coherent clusters.

However this merging step would probably need $O(n^2)$ operations.

In cases where no connectivity information is present, it may be possible to use an enhanced version of the approach discussed in Section 3.7. At the end of this section, the problem arising from clusters of certain shapes have been described. These problems arise when a cluster is wide stretched in one dimension but rather small in another dimension. This is incompatible with conventional interpolation schemes. It might be possible to modify the interpolation scheme and gain satisfactory results. *E.g.*, it might be possible, to use a different distance measure when computing the interpolation value. A possibility is to use principal component analysis (PCA) (see [20, 21, 26]) to find the inherent coordinate system of the clusters. Given a set of points, here the positions of the samples of a given cluster, PCA yields a coordinate system, which has axises in direction of maximum and minimum extent of the point set. This coordinate system can be used to define a metric, that reflects the extent of the cluster, *i.e.*, equidistant points from the cluster representant are located on an ellipsoid instead of a sphere, and yields correct interpolation results.

Furthermore, the clustering process can be integrated into the visualization system. If each leaf node in the clustering tree contains a reference to the samples in the original data set that are within the corresponding cluster, it is possible to refine the hierarchy beyond the pre-computed levels. Whenever a user chooses a region that cannot be satisfactorily approximated by the given hierarchy, it is possible to refine it. This can be done by splitting the clusters corresponding to the leaf nodes within the ROI. If these clusters are sufficiently small, this can be done on the fly during the visualization. Thus it becomes possible to represent the data set at arbitrary levels of detail.

6.3 Improving procedural mesh generation process

The procedural generation of a local simplical mesh can be improved in several ways. First, the Delaunay triangulation can be replaced with a different triangulation or mesh generation method. Section 2.5 stated that in the 2D case the Delaunay triangulation is the optimal triangulation for the vector field. This is not true in 3D space. The Delaunay triangulation stills yield a high-quality triangulation and is widely used for mesh generation, but it is not the only method. It might prove favorable to replace the Delaunay triangulation with a triangulation scheme with less computational cost, even if the there is a slight decrease in quality of the resulting triangulation.

Using a less expensive triangulation method could also make the first approaches discussed in Section 4.3 feasible. Using triangulation routines which allow the dynamic insertion and removal of points in the triangulation would also open new possibilities. It may become possible, to extent the current triangulation toward the queried point instead of generating a completely new triangulation, when a query for a position outside the current triangulation is performed. This could be done by adding points in the region of the queried point to the triangulation while removing positions from the "opposite side" of the triangulation.

A possible extension to the lastly discussed approach which utilizes the hierarchy is also possible. Currently the complete definition region of the vector field is refined, until the desired

amount of samples is within the region of interest. Time and memory is spent in recreating precomputed splits that do not have any impact on the region of interest. An improved approach could check whether a node has descendants in the region of interest. If this is not the case, the split described by this node could be ignored. This can be done by, *e.g.*, computing a bounding box of the samples of the original data set that are represented by a given node and intersecting it with the ROI. This approach could also eliminate the need of inserting the vertices of the region of interest in the triangulation. If in the first step an approximation for the entire vector field is found and this is only refined in the region of interest this results in a triangulation of the complete vector field. The number of positions is still sufficiently small, since only the region of interest is represented in finer detail.

6.4 A fully functional visualization system for the Immersive WorkBench

The current implementation of the visualization on the Immersive WorkBench is a prototype that has bee was used to examine approaches in programming the workbench and using it to visualize vector fields. The possibilities for improvements are multifarious.

The current menus, *i.e.*, the watch menu should be replaced by a more powerful version of it. Alternatively, it would be possible to use an existing Widget set for the workbench, like Multigen SmartScene[1].

Furthermore, the current mixture of OpenGL and OpenInventor should be avoided. Currently, the best choice choice for an API to program the Immersive WorkBench is *IRIS Performer* by Silicon Graphics. This library is recommended by Silicon Graphics for real-time graphics applications like VR and VE. Like OpenInventor it offers a scene graph based API and is even able to read OpenInventor geometry files. The use of IRIS Performer was considered in the course of this work. Unfortunately, this approach had to be abandoned because the documentation for the part of Fakespace VLIB allowing to develop workbench applications based on IRIS Performer described a different version of this library and provided insufficient information for an actual implementation. Otherwise, a scene graph based API could have eliminated the need of global display lists and transformation information. In the case of a scene graph it would be sufficient to share a simple pointer to the root node between “tracker thread” and “display thread.”

The current implementation provides an event-based handling of the stylus data. A logical step is to implement this also for the pinch gloves. This would eliminate the need of modifying the transformation of the model-on-a-stick coordinate system in the function `trackerThread()`. This function should only query the sensor data and translate it into events. The rest should be done in other modules.

The currently implemented visualization objects are only the beginning in the implementation of a complete visualization system. It should be possible to map scalar values as color

[1] Unfortunately the version of Multigen SmartScene purchased by CIPIC was not available soon enough to be used in this Diplomarbeit.

codes on them. Furthermore additional visualization objects should be added.

One problem arising in this is the difficulty of entering numerical values. At the moment, it is only possible to get a qualitative impression of the vector field. A complete visualization should enable the user to get a fully quantitative view of the vector field. This demands a powerful mechanism to enter numerical data in a VE, which is an interesting problem on its own.

Appendix A

Libraries and implementation

For the implementations within the scope of this Diplomarbeit the following libraries and implementations were used.

A.1 General data structures and fundamentals

- Standard C/C++ libaries (I/O, basic data structures, etc.)
- Standard Unix libraries (Semaphores, I/O, etc.)

A.2 Graphical User Interface (GUI) and Graphics

- X11R6: XLib, X Intrinsics by the X Consortium
- Motif by the Open Software Foundation (OSF).
- OpenGL by Silicon Graphics, Mt. View, CA, USA.
- OpenInventor by Silicon Graphics, Mt. View, CA, USA.
- RViewer class by René Schätzl (OpenInventor "`SoXtExaminerViewer`" and Motif Widgets in one window)
- VLIB (for using the Immersive WorkBench) by Fakespace, Mt. View, CA, USA.
- OpenGL helper library by Oliver Kreylos (matrix operations, geometric primitives, etc.)

A.3 Computational geometry

- LEDA (2D Delaunay Graph) by MPI Saarbrücken, Germany.

- ANN (k-D-tree) written by David Mount, Dept. of Computer Science, University of Maryland, College Park, MD 20742, USA.
- 3D Delaunay triangulation code from Dave Eberly, Dept. of Computer Science, University of North Carolina at Chapel Hill, USA.
- hull (3D delaunay triangulation via convex hull) by K. L. Clarkson, Bell Laboratories, Murray Hill, NJ 07974-0636, USA.

A.4 Miscellaneous

- Basic Clustering sources by Björn Heckel, Dept. of Computer Science, University of California at Davis, Davis, CA 95616, USA.
- Hardy Interpolation sources by Björn Heckel, Dept. of Computer Science, University of California at Davis, Davis, CA 95616, USA.
- MESCHACH (C routines for performing calculations on matrices and vectors) written by D.E. Stewart, and Z. Leyk, School of Mathematical Sciences, Australian National University, Canberra ACT 0200, Australia

Appendix B

Used resources

B.1 Software packages

- Typesetting was done using LaTeXby Donald E. Knuth with the teTeXpackage by Thomas Esser, Universität Hannover, Germany.
- Most of the Figures were drawn using the `xfig` package written by Supoj Sutanthavibul at the University of Texas at Austin and Brian V. Smith.
- Conversions of Plot 3D data sets were done using the FAST package by NASA Ames Research Center (`http://science.nas.nasa.gov/Software/FAST/RND-93-010.walatka-clucas/htmldocs/titlepage.html`).

B.2 Figures

- Figure 2.1 is closely based on Figure 5.3 in [2].
- Figures 2.16 and 2.19 are taken from the web page "Virtual Reality: A Short Introduction" (`http://www-vrl.umich.edu/intro.html`) by K.-P. Beier, Virtual Reality Laboratory (VRL), College of Engineering, University of Michigan.
- Figure 2.18 is courtesy of Falko Küster, CIPIC, Dept. of Computer science, Univeristy of Califrnia at Davis, CA 95616, USA.
- Figure 2.20 was taken from Stanford Computer Graphics Laboratory Research Projects homepage (`http://graphics.stanford.edu/projects/RWB/bench.jpg`).
- The plane used in Figures 2.21, 2.22, and 2.23 was rendered from the file `X29.iv` supplied by Silicon Graphics, Mt. View, CA, USA in conjunction with OpenInventor using the supplied command line tool `ivview`.

B.3 Data sets

- The blunt fin dataset used to render Figures 2.14, 2.15, 5.12(a), 5.12(b), 5.12(c), 5.13(a), 5.13(b), 5.13(c), 5.14, 5.15, 5.16, and 5.17 is courtesy of NASA Ames and can be found at `http://chuck.nas.nasa.gov/DataSets/Hung.1/`

Bibliography

[1] BENTLEY, J. L. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (Sept. 1975), 509–517.

[2] BERG, M., KREVELD, M., OVERMARS, M., AND SCHWARZKOPF, O. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Heidelberg, New York, 1997.

[3] BRYSON, S., AND LEVIT, C. The virtual windtunnel: An environment for the exploration of three dimensional unsteady flows. Tech. Rep. RNR-92-013, NAS Technical Reports, Apr. 1992.

[4] CIGNONI, P., DE FLORIANI, L., MONTONI, C., PUPPO, E., AND SCOPIGNO, R. Multiresolution modeling and visualization of volume data based on simplicial complexes. In *1994 Symposium on Volume Visualization* (Oct. 1994), A. Kaufman and W. Krueger, Eds., ACM SIGGRAPH, pp. 19–26.

[5] CIGNONI, P., MONTANI, C., PUPPO, E., AND SCOPIGNO, R. Multiresolution representation and visualization of volume data. *IEEE Transactions on Visualization and Computer Graphics 3*, 4 (1997), 352–369.

[6] FOLEY, J. D., VAN DAM, A., FEINER, S. K., AND HUGHES, J. F. *Computer Graphics, Principles and Practice, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1990.

[7] FOLEY, T. A., AND HAGEN, H. Advances in scattered data interpolation. *Surveys on Mathematics for Industry 4*, 2 (1994), 71–84.

[8] FRIEDMAN, J. H., BENTLEY, J. L., AND FINKEL, R. A. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Transactions on Mathematical Software 3*, 3 (Sept. 1977), 209–226.

[9] FRÜHAUF, T. *Graphisch-Interaktive Strömungsvisualisierung*. Springer Verlag, Berlin, Heidelberg, 1996. Zugl.: Darmstadt, Techn. Hochsch., Diss., 1996.

[10] GIENG, T. S., HAMANN, B., JOY, K. I., SCHUSSMAN, G. L., AND TROTTS, I. J. Constructing hierarchies for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics 4*, 2 (1998), pp. 145–161.

[11] GORDON, A. D. Hierarchical classification. In *Clustering and Classification*, R. Arabie, L. Hubert, and G. DeSoete, Eds. World Scientific Publishers, River Edge, NJ, 1996, pp. 65–105.

[12] HARDY, R. L. Multiquadric equations of topography and other irregular surfaces. *Journal of Geophysical Research 76* (1971), 1906–1915.

[13] HARDY, R. L. Theory and applications of the multiquadric-biharmonic method: 20 years of discovery 1968–1988. *Computers and Mathematics with Applications 19* (1990), 163–208.

[14] HECKEL, B., AND HAMANN, B. Visualization of cluster hierarchies. In *Proceedings of the SPIE - The International Society for Optical Engineering* (Jan. 1998), vol. 3298, pp. 162–171.

[15] HECKEL, B., UVA, A. E., AND HAMANN, B. Clustering-based generation of hierarchical surface models. In *Proceedings of Visualization 1998 (Hot Topics)* (Oct. 1998), C. Wittenbrink and A. Varshney, Eds., IEEE Computer Society Press, Los Alamitos, CA, pp. 50–55.

[16] HECKEL, B., WEBER, G. H., HAMANN, B., AND JOY, K. Construction of vector field hierarchies. Submitted paper for IEEE Visualization '99.

[17] HEINRICH, S. Numerische Mathematik für Informatiker. Skriptum zur Vorlesung "Numerik" an der Universiät Kaiserslautern.

[18] HOPPE, H. Progressive meshes. In *SIGGRAPH 96 Conference Proceedings* (Aug. 1996), H. Rushmeier, Ed., Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 99–108.

[19] HOROWITZ, E., SAHNI, S., AND MEHTA, D. *Fundamentals of Data Structures in C++*. Computer Science Press, 41 Madison Avenue, New York NY 10010 USA, 20 Beaumont Street, Oxford OX1 2NQ England, 1995.

[20] HOTELLING, H. Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology 24* (1933), pp. 417–441 and 498–520.

[21] JACKSON, J. E. *A User's Guide to Principal Components*. Wiley, New York, 1991.

[22] KENWRIGHT, D. N. *Dual stream function methods for generating three-dimensional stream lines*. PhD thesis, Department of Mechanical Engineering, University of Auckland, Aug. 1993.

[23] KNIGHT, D., AND MALLINSON, G. Visualizing unstructured flow data uisng dual stream functions. *IEEE Transactions On Visualization And Computer Grpahics 2*, 4 (Dec. 1996).

[24] KREYLOS, O. Optimale Approximation uni- oder multivariater Funktionen in mehreren Auflösungen, 1998. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe.

[25] KREYLOS, O., AND HAMANN, B. On simulated annealing and the construction of linear spline approximations for scattered data. In *Proceedings EUROGRAPHICS—IEEE TCCG Symposium Visualization* (Vienne, Austria, May 1999).

[26] MANLY, B. *Multivariate Statistical Methods, A Primer*. Chapman & Hall, New York, New York, 1994.

[27] MOUNT, D. M. Ann programming manual, 1999. Manual for ANN libarary, `http://www.cs.umd.edu/~mount/ANN/`.

[28] MÜCKE, E. P., SAIAS, I., AND ZHU, B. Fast randomized point location without preprocessing in two-and three-dimensional Delaunay triangulations. In *12th ACM Symp. on Computational Geometry* (May 1996). `ftp://ftp.cfar.umd.edu/TRs/CVL-Reports-1996/TR3621-Mucke.ps.gz`.

[29] NEHMER, J. Betriebsorganisation von Rechnersystemen: Eine Einführung auf Basis des Client/Server-Modells. Skriptum zur Vorlesung "Betriebssyteme" an der Universiät Kaiserslautern.

[30] NIELSON, G. M., HAGEN, H., AND MÜLLER, H., Eds. *Scientific Visualization: Overviews, Methodologies, Techniques*. IEEE Computer Society, Los Alamitos, California, 1997.

[31] NIELSON, G. M., JUNG, I.-H., AND SUNG, J. Haar wavelets over triangular domains with applications to multiresolution models for flow over a sphere. In *IEEE Visualization '97* (Los Alamitos, Oct. 1997), R. Yagel and H. Hagen, Eds., IEEE Computer Society Press, pp. 143–149.

[32] NIELSON, G. M., JUNG, I.-H., AND SUNG, J. Wavelets over curvilinear grids. In *Proceedings IEEE Visualization'98* (Los Alamitos, 1998), D. S. Ebert, H. Hagen, and H. Rushmeier, Eds., IEEE Computer Society Press, Los Alamitos, CA, pp. 313–317.

[33] OKABE, A., BOOTS, B., AND SUGIHARA, K. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. Probability and Mathematical Statistics. John Wiley & Sons, Chichester, England, Sept. 1992. foreword by D. G. Kendall.

[34] O'ROURKE, J. comp.graphics.algorithms Frequently Asked Questions. `http://www.exaflop.org/docs/cgafaq/`.

[35] PREPARATA, F. P., AND SHAMOS, M. I. *Computational Geometry: In Introduction*. Springer-Verlag, New York, 1985.

[36] PRESS, W. H., FLANNERY, B. P., TEUKOLSKY, S. A., AND VETTERLING, W. T. *Numerical Recipies in C: The art of Scientific Programming*. Cambridge University Press, Cambridge, England, 1988.

[37] SCHROEDER, W., MARTIN, K., AND LORENSEN, B. *The Visualization Toolkit: An object oriented approach to 3D graphics*. Prentice Hall, 1996.

[38] SCHROEDER, W. J., ZARGE, J. A., AND LORENSEN, W. E. Decimation of triangle meshes. In *Computer Graphics (SIGGRAPH '92 Proceedings)* (July 1992), E. E. Catmull, Ed., vol. 26, pp. 65–70.

[39] SEDGEWICK, R. *Algorithms in C++*. Addison-Wesley, Reading, MA, USA, 1992.

[40] STAADT, O. G., AND GROSS, M. H. Progressive tetrahedralizations. In *Proceedings of Visualization 98* (Oct. 1998), D. S. Ebert, H. Hagen, and H. Rushmeier, Eds., IEEE Computer Society Press, Los Alamitos, California, pp. 397–402.

[41] STOLLNITZ, E. J., DEROSE, T. D., AND SALESIN, D. H. Wavelets for computer graphics: A primer, part 1. *IEEE Computer Graphics and Applications 15*, 3 (May 1995), 76–84.

[42] STOLLNITZ, E. J., DEROSE, T. D., AND SALESIN, D. H. Wavelets for computer graphics: A primer, part 2. *IEEE Computer Graphics and Applications 15*, 4 (July 1995), 75–85.

[43] TROTTS, I. J., HAMANN, B., JOY, K. I., AND WILEY, D. F. Simplification of tetrahedral meshes. In *Proceedings of Visualization 98* (Oct. 1998), D. S. Ebert, H. Hagen, and H. Rushmeier, Eds., IEEE Computer Society Press, Los Alamitos, CA, pp. 287–296.

[44] WOO, M., NEIDER, J., DAVIS, T., AND OPENGL ARCHITECTURE REVIEW BOARD. *OpenGL programming guide: the official guide to learning OpenGL, version 1.1*. Addison-Wesley, Reading, MA, USA, 1997.