JACOBS
UNIVERSITY

Direct Surface Extraction from Unstructured Point-based Volume Data

Paul Rosenthal

A thesis submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
in
Computer Science

Approved, Dissertation Committee:
Prof. Dr.-Ing. Lars Linsen (supervisor)
Prof. Dr. Stephan Rosswog
Prof. Dr.-Ing. Horst K. Hahn
Prof. Dr. Daniel Weiskopf

Date of Defense: April 30, 2009
School of Engineering and Science
Jacobs University, Bremen, Germany

To Sabine

Executive Summary

Surface extraction is a standard visualization method for scalar volume data and has been subject to research for decades. Many algorithms for surface extraction from various data structures and types exist. However, for unstructured point-based volume data, where no topology or connectivity between data points is given, most approaches propose to reconstruct the scalar field over a grid and apply standard surface extraction techniques for the obtained grids. This work introduces a new method that directly extracts surfaces from unstructured volume data without three-dimensional mesh generation or reconstruction over a structured grid. The presented approach consists of two major processing steps: a geometry extraction step and a point-cloud rendering step.

The geometry extraction step computes points on the isosurface by linearly interpolating between neighboring pairs of samples. The needed neighbor information is retrieved by approximating natural neighbors as provided by Voronoi diagrams. One presented approximation approach is the generation of a three-dimensional discrete Voronoi diagram. This is done with the aid of today's graphics hardware to achieve reasonably fast results. A second approach for approximating natural neighbors is partitioning the three-dimensional domain into cells using a kd-tree. The cells are merely described by their index and bitwise index operations allow for a fast determination of potential neighbors. An angle criterion is used to select appropriate neighbors from the small set of candidates. The approach is evaluated on several synthetic data sets and is significantly faster than previously developed algorithms while assuring nearly the same accuracy. Moreover, it is much more accurate than reconstructing the scalar field over a regular grid.

In sparsely sampled regions, extracted isosurfaces could be rough especially when dealing with noisy data. To avoid such issues, a level-set approach can be applied to the data before isosurface extraction. This results in smoother segmenting surfaces when extracting the geometry of the zero level set. In contrast to existing level-set approaches, which operate on gridded data and mainly on regular structured grids, an approach is presented that directly computes level sets on unstructured point-based volume data without prior resampling or mesh generation. To suffice the needs of smooth segmentation regarding an isovalue, a level-set method is chosen that combines hyperbolic advection to the isovalue and mean curvature flow. The needed function properties like gradient and mean curvature are approximated in each sample point by a consistent least-squares approach operating in four dimensions. Since the approach uses an explicit time-integration scheme, time steps are bounded by the Courant-Friedrichs-Lewy condition. To avoid small global time steps, asynchronous local integration can be applied. The practicality of this approach is shown on simulated smoothed particle hydrodynamics data.

The output of the geometry extraction step is a point-cloud representation of the isosurface, where each point only holds its position information and a vector inducing the surface orientation. A point cloud containing surface normal information is generated using a least-squares approach. The final rendering step uses point-based rendering techniques to visualize the point cloud. If a fast and interactive rendering is needed, an algorithm based on image-space operations is used. The lit point cloud is directly rendered to screen space, possibly resulting in holes in the rendered surface. These holes are detected using image-space filters and filled with neighboring surface color information. A final smoothing filter assures a smooth surface rendering. If the rendering should include photorealistic effects, a ray-tracing approach is preferable. The surface is approximated using circular splats with attached normal field. These splats are then rendered with ray tracing to achieve photorealism.

The presented direct surface extraction algorithm for unstructured point-based volume data produces results of high quality. By applying the level-set approach in a preprocessing phase, it allows for a smooth yet correct surface extraction also for data sets with highly varying point density. Nevertheless, the proposed methods are competitive with similar powerful approaches in terms of computation speed.

Acknowledgments

First, I want to express my sincere gratitude to my thesis supervisor Lars Linsen. He was the man, turning me, a pure mathematician addicted to computer games, to a computer scientist actually able to write this thesis. I thank him for accepting me as doctoral candidate and introducing me into the exciting and diversified field of visualization. Thank you also, for the various valuable discussions and the guidance through numerous papers and finally this thesis.

My special thanks go to my dearest, Sabine. Without her, I would have never studied mathematics and would have never discovered the wonderful world of natural sciences. She always supported me in good and in bad times, tried to understand and follow my sometimes confusing ideas, traveled with me to numerous conferences, always listened to my problems – or solutions, and finally read this thesis several times.

A special gratitude goes also to Stephan Rosswog, for making my research useful by providing his data sets, for various discussions, and for valuable input from a different point of view. In this context, I also want to thank Horst Hahn and Daniel Weiskopf for reviewing this thesis and my colleagues Tatyana Ivanovska, Steffen Brasch, Petar Dobrev, Steffen Hauth, and Tran Van Long for their productive collaboration.

Furthermore, I want to thank the Jacobs University in Bremen for facilitating this Ph.D. project and offering so nice study conditions. Finally, I thank my whole family for supporting me materially and ideally at all times.

Contents

Chapter 1

Introduction

Everything is vague to a degree you do not realize till you have tried to make it precise.

Bertrand Russell

The modern world is ruled by strictness and regularity. Not only that nearly all parts of life are legislated. With the rise of information technology also the relation between information and people has dramatically changed. Most information is nowadays stored or transmitted as data, i. e. discretized information, in contrast to the past, when most of the information has been either "stored" as human knowledge or scriptures. Also the amount of produced data increases more and more. Nevertheless, it is indispensable not only for scientists to gain insight into the data to draw conclusions or find results. Hence, it is necessary to provide techniques to make the huge amounts of data understandable for human mind. This procedure of reprocessing data into human understandable visual models is the main goal of visualization.

Since all objects in our world are volumetric, the visualization of volumetric data has become a major direction over the past years. And as most of the generated volumetric data sets are gridded, i. e. there exist connectivity informations of the data points, the great majority of visualization techniques deal with all different types of gridded data. The most commonly used visualization technique beside direct volume rendering [DCH88] is the extraction of two-dimensional surfaces with specific properties in data domain. The simplest definition of such a surface is an isosurface with respect to an isovalue $f_{\text{iso}} \in \mathbb{R}$. For a data set representing the scalar field $f : \mathbb{R}^3 \to \mathbb{R}$ an isosurface is the submanifold $\Gamma_{\text{iso}} \subset \mathbb{R}^3$, defined by

$$\Gamma_{\text{iso}} := \left\{ \mathbf{x} \in \mathbb{R}^3 \ : \ f(\mathbf{x}) = f_{\text{iso}} \right\} \ .$$

If the points on the surface have to fulfill different properties, these are typically encoded into an auxiliary scalar field, such that the desired surface is again just an isosurface to the constructed field. Hence, nearly every surface extraction problem

can be reduced to the problem of isosurface extraction and there exists a vast variety of different isosurface extraction algorithms for all different types of grids [CHJ03, CPJ04, KKDH07, LC87, Pas04, SSS06, WKL+03].

However, many modern measuring methods for environmental parameters, like for example smart dust [WLLP01] or ocean sensors [RO99], generate data that is not gridded anymore. These provide new insights by additional degrees of freedom, i. e. by movement of the measuring instruments themselves. Also many modern simulation approaches modeling physical processes utilize these additional degrees of freedom. Mesh-free Lagrangian methods like smoothed particle hydrodynamics [Luc77, GM77] not only simulate the evolution of the data at the sample points but also simulate the flow of the sample points under respective forces.

In particular, the smoothed particle hydrodynamics method is a method simulating physical flows. The object of interest, e. g. a fluid or an astrophysical object like a star, is represented by a discrete set of particles each with a distinct dimension, over which the objects properties are smoothed by a kernel function. With this representation of the object the transformation regarding partial differential equations, modeling pressures and other forces, is done using a Lagrangian particle method. More precisely not only the properties like temperature, density, and mass fractions are affected by the partial differential equations, but also the particle positions. In every point in time of the simulation the output is an unstructured point-based data set.

Although so many isosurface extraction approaches have been introduced for various data structures, there existed no algorithm that directly operates on unstructured point-based volume data, where no connectivity between the data points is known. To deal with this type of data, the data was typically resampled over a regular structured grid using scattered data interpolation techniques or converted into an irregular grid using a polyhedrization technique. While the former approach may produce resampling inaccuracies, the latter is typically computationally expensive and often rather cumbersome to implement.

In this thesis, a complete visualization pipeline for direct surface extraction from unstructured point-based volume data is presented, i. e. we are not resampling the data over a structured grid nor are we generating global or local polyhedrizations. An illustration of the visualization pipeline is shown in Figure 1.1.

Starting from an unstructured point-based scalar data set, isosurfaces can be extracted using direct isosurface extraction [RL06]. Points on the isosurface are linearly interpolated between neighboring sample points. For obtaining these neighbors two different approaches are introduced, both approximating natural neighbors as defined by Voronoi diagrams. The former technique utilizes modern graphics hardware to construct discrete Voronoi diagrams to obtain natural neighbors, while the latter method uses fast neighbor approximations, based on an indexing scheme of a space partitioning kd-tree. A comprehensive description and comparison in terms of quality and computation time of both isosurface extraction methods is given in Chapter 2.

The high flexibility makes unstructured point-based data very powerful and enables

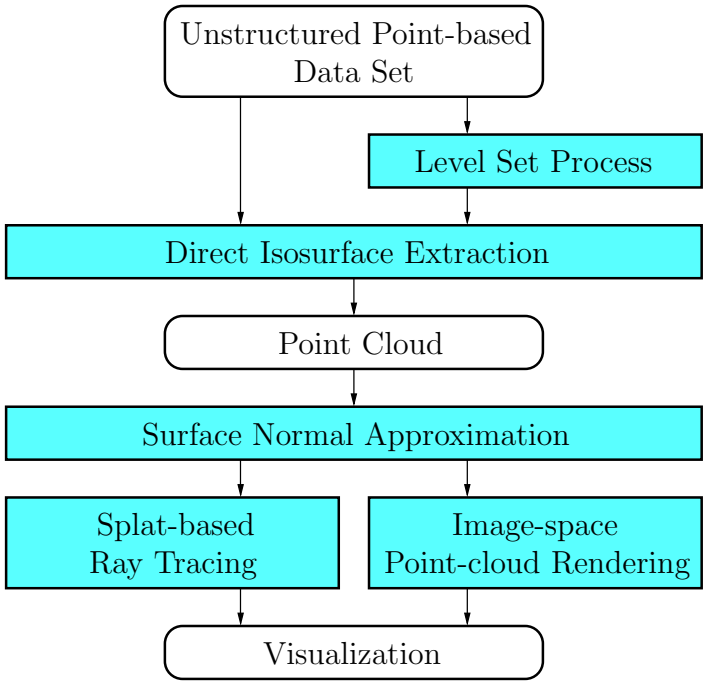


Figure 1.1: Pipeline for direct surface extraction from unstructured point-based volume data.

the modeling of complex relations. However, especially this flexibility raises problems for direct isosurface extraction. For data sets with highly varying sample-point density, as often generated by astrophysical simulations, the linear interpolation between far apart sample points can lead to inaccuracies in the isopoint computation. To overcome this possible problem, we propose a level-set-based preprocessing step that again directly operates on the unstructured point-based volume data set [RL08b]. An auxiliary level-set function is introduced at the sample point locations. The level-set function is deformed following a partial differential equation combining hyperbolic advection and mean curvature flow such that it stays smooth and the zero level set approximates the desired isosurface. Spatial derivatives of the level-set function, which are needed for the level-set process, are directly approximated using least-squares methods. From this level-set function the desired isosurface can be extracted, also in noisy data sets or data sets with highly varying point densities. In Chapter 3 we present the theoretical considerations as well as practical examples and results of the proposed level-set method.

Since the unstructured point-based data is never resampled and no connectivity is known, the resulting isosurface is generated in point-cloud representation. More precisely a set of isopoints is calculated which lie on the isosurface but exhibit no connectivity information. To avoid time-consuming surface reconstruction steps, the resulting point cloud has to be visualized using point-based rendering techniques. We first approximate surface normals at the surface points with a least-squares approach. Afterwards we open up two possibilities for point-cloud rendering, either splat-based ray tracing or image-space point-cloud rendering.

To generate high-quality renderings of the surface with photorealistic effects like global illumination, reflection, and refraction, a splat-based ray-tracing approach [LMR07] is presented. Here the surface is locally approximated by small discs (splats) with attached normal fields and the resulting set of splats is ray traced. This approach needs some precomputations and is not able to generate interactive visualizations.

If an interactive visualization without precomputations is needed, we propose an image-space point-cloud rendering process [RL08a], generating high-quality and smooth surface renderings at interactive frame rates without any precomputations. This method uses modern graphics hardware capabilities to generate point-cloud renderings. The lit point cloud is projected to screen space. Possible holes in the surface are filled with image-space filters. The optional processing of an additional normal map opens up the possibility of applying anti-aliasing or illustrative rendering techniques. Both point-cloud rendering techniques are explained in detail in Chapter 4.

The proposed visualization pipeline for surface extraction from unstructured point-based volume data is tested with the help of several well-known data sets as well as data sets directly provided from the above-mentioned application areas. The results are assessed in terms of quality and speed. Additionally they are compared to similar approaches where possible.

Chapter 2

Direct Isosurface Extraction

Isosurface extraction is the most commonly used visualization technique for scalar volume data and many isosurface extraction algorithms exist. Although all these approaches operate on various data structures, no algorithm existed that directly operated on unstructured point-based volume data, where no grid connectivity is given.

The most common way of dealing with this type of data is to to resample the data to a structured grid using scattered data interpolation techniques [FN91]. Scattered data interpolation is a well-studied field and Park et al. [PLK+06] have shown that scattered data reconstruction for large data sets can be achieved at interactive or near-interactive rates when resampling over a regular grid. Unfortunately, such resampling steps always induce inaccuracies, which can grow enormously for data sets with highly varying point densities.

An alternative would be the calculation of a tetrahedral grid from the unstructured point-based data [Nie93]. Delaunay tetrahedralizations [Del34] are known to produce desirable results. Since the asymptotic complexity for the tetrahedralization is quadratic, it is very time consuming and not practicable for large sets of sample points. Actually applying the standard tetrahedralization algorithm provided with the Computational Geometry Algorithms Library [CGA] to a data set with four million sample points takes 54 minutes of computation time.

Another well-known approach for visualizing scalar volume data is direct volume rendering [Lev88]. This method is able to produce renderings of isosurfaces directly from the volumetric data by tracing view rays. Originally proposed for different types of grids, recently several volume rendering approaches for unstructured point-based volume data [HE03, HLE04, TSE07] have been proposed. However, the used scattered data interpolation techniques still introduce interpolation errors. For direct volume rendering of smoothed particle hydrodynamics data, the visualization tool SPLASH [Pri07] was proposed, which utilizes the smoothed particle hydrodynamics kernel to interpolate along the view rays. However, another advantage for isosurface extraction is the actual extraction of the isosurfaces' geometry, which can be stored and used for further research.

Recently, Co et al. [CPJ04, CJ05] have presented an algorithm for isosurface extraction from unstructured point-based volume data. However, instead of generating the points on the isosurface directly from the unstructured volume data, they generate local tetrahedral grids and compute the isopoints from the local grid. Nevertheless, this approach is the only one that allows unstructured point-based data as input and we will compare our approach to this in Section 2.5 in detail.

In this chapter we describe two methods for direct isosurface extraction from unstructured point-based volume data, the first based on discrete Voronoi diagrams and the second kd-tree-based. Points on the isosurface are computed by interpolating between neighboring samples of the volume data set. The inspiration for this technique comes from the marching tetrahedra algorithm [BCL06, Pas04, TPG99], where a given tetrahedral structure is used for linear interpolation. Since no connectivity between sample points is given, a neighbor relation has to be calculated. The use of nearest neighbors [Cle79, Ben80, BF79, FBF77] is not suitable to compute the pairs of neighboring points especially for highly varying point densities.

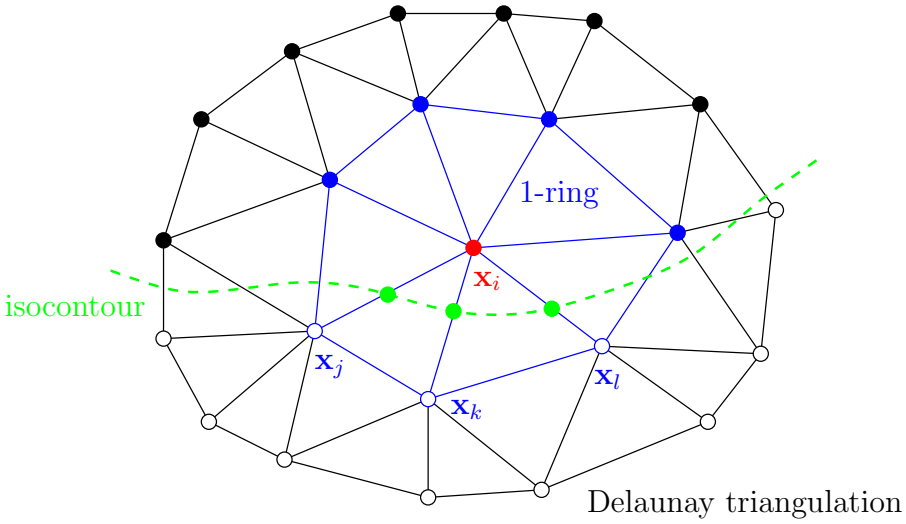


Figure 2.1: Isopoint computation for unstructured point-based data via Delaunay triangulation and linear interpolation between natural neighbors, i. e. members of the 1-ring of a point in the triangulation. For sample position $\mathbf{x}_i$, the outgoing edges to $\mathbf{x}_j$, $\mathbf{x}_k$, and $\mathbf{x}_l$ intersect the isocontour.

The best choice for the neighbor relation would be natural neighbors induced by the Delaunay tetrahedralization, as illustrated in Figure 2.1. However, the calculation, which typically implies the calculation of a Voronoi diagram, is not practicable for large data sets. That is why a method for obtaining an approximation for the natural neighbors of each sample point is needed. Two different approaches are presented in the following.

The first approach, described in Section 2.1, does not directly approximate natural neighbors for each sample point but generates a discrete Voronoi diagram, i. e. an approximation for the Voronoi diagram of the sample points. From this discrete

Voronoi diagram, a set of natural neighbor candidates is computed for each sample point.

A direct approximation of the natural neighbors for each sample point is computed by the second approach, described in Section 2.2. It uses a spatial domain decomposition based on a three-dimensional kd-tree and a fast neighbor search based on an efficient indexing scheme and bitwise operations on the tree indices.

Both algorithms produce sets of neighbor candidates for each sample point. However, these neighbor candidates only assure a complete surrounding of each sample with neighbors. A second property of natural neighbors is a distribution around the point which is proportional to the relative point density. To additionally assure this property, an angle criterion has to be applied to the neighbor candidates to produce the final approximation of the natural neighborhood for each sample point. The algorithm restricting the angles between neighboring points is described in detail in Section 2.3.

The actual isopoint computation using linear interpolation between neighboring sample points is described in Section 2.4. A comprehensive overview of the obtained results of both algorithms is given in Section 2.5. Advantages and limitations of the different approaches are compared and discussed with respect to the actual data and needs.

Finally, additional applications for the proposed isosurface extraction techniques, arising in the wide field of multi-dimensional data visualization, are given. In Section 2.6 we show how the proposed methods can be used to visualize clusters, obtained from multi-dimensional and multi-variate data.

2.1 Discrete Voronoi Diagram Calculation

For the extraction of reasonable isopoints, a good approximation of the natural neighborhood of each sample point is needed. The computation of a natural neighborhood would typically base on a Voronoi diagram [Aur91] of the sample points. However, the exact computation of a three-dimensional Voronoi diagram of n points has a complexity of $O(n^2)$ [Ede87] and is not applicable to big sets of points.

Hence, an approximation of the Voronoi diagram of the sample points is needed. Several algorithms for the approximation of Voronoi diagrams like (t, ε)-approximate Voronoi diagrams [AMM02] or Voronoi Octrees [BCMS08] exist. Here the approximation by a discrete Voronoi diagram is chosen due to the simple definition and fast calculation.

Definition 2.1 *Let $M \subset D$ be a set of unstructured sample points in the domain $D \subset \mathbb{R}^3$. A uniform three-dimensional hexahedral grid G on D is a disjoint covering of D with equal cubes. These cubes are called grid cells and such a grid will be denoted regular grid.*

A regular grid induces a canonical mapping $\psi : M \to G$ of each sample point $\mathbf{x} \in M$ to one cell of the grid.

With the introduction of a regular grid G on the domain of the sample points we define a distance function $d_G : G \times G \to \mathbb{R}_0^+$ induced by the Euclidian distance d by

$$d_G : (g_1, g_2) := d\,(p_1, p_2)\ ,$$

where the point p_i denotes the barycenter of the cell g_i for $i \in \{1, 2\}$.

Definition 2.2 *Let $M \subset D$ be a set of unstructured sample points in $D \subset \mathbb{R}^3$ and G be a regular grid on D. The discrete Voronoi diagram to the set of points M with respect to G is the function $\psi_{\mathrm{V}} : G \to \psi(M)$, mapping each grid cell of G to the nearest filled grid cell with respect to d_G.*

Due to the variety of applications of Voronoi diagrams, the idea of approximating them with discrete Voronoi diagrams is not new and several approaches exist [TT97, MRH00]. With the rise of general purpose computations on graphics cards several algorithms using the features of GPUs were introduced [SPG03, SGM05, HT05].

The presented approach for creating a discrete Voronoi diagram for the sample points is based on the ideas by Hoff et al. [HCK+99, HKL+99]. They create two-dimensional discrete Voronoi diagrams by rendering cones and using the depth buffer of the graphics card. In one rendering step, a cone for each sample point is rendered to a texture, resulting in a discrete Voronoi diagram. From this texture the natural neighbors for each of the sample points can be easily derived.

The authors of both papers also describe ideas for remodeling the approach to generate three-dimensional discrete Voronoi diagrams. This should be done in two-dimensional layers, as it is not possible to render into three-dimensional textures. However, as they state in their papers, they are just using a simple brute-force strategy to calculate the three-dimensional discrete Voronoi diagrams instead of actually finalizing the ideas and implementing them to show the practicability.

This adoption of the ideas to three dimensions is indeed possible and is in the presented approach used for generating neighbor candidates. The three-dimensional discrete Voronoi diagram is generated in two-dimensional layers. In contrast to the algorithm for two dimensions, points not lying in the actual layer can generate Voronoi cells there. Hence, all sample points have to be considered for each layer of the discrete Voronoi diagram.

Moreover, the real three-dimensional distances between points from different layers do not linearly depend on the two-dimensional distance of the projected points, as illustrated in Figure 2.2. For this reason, it is not possible to render just cones to generate the discrete Voronoi diagram. More precisely we also have to use conic sections, which is explained in more detail in the following.

Following the ideas by Hoff et al., one would need to render a three-dimensional cone

$$C_{\mathbf{x}} := \left\{(y_1, y_2, y_3, y_4) \in \mathbb{R}^4\ :\ |(y_1, y_2, y_3) - \mathbf{x}| = y_4\right\}\ ,$$

represented as a subspace of $\mathbb{R}^4$ for each sample point $\mathbf{x} \in \mathbb{R}^3$. These cones would have to be rendered to $\mathbb{R}^3$ using depth testing, resulting in a three-dimensional discrete Voronoi diagram.

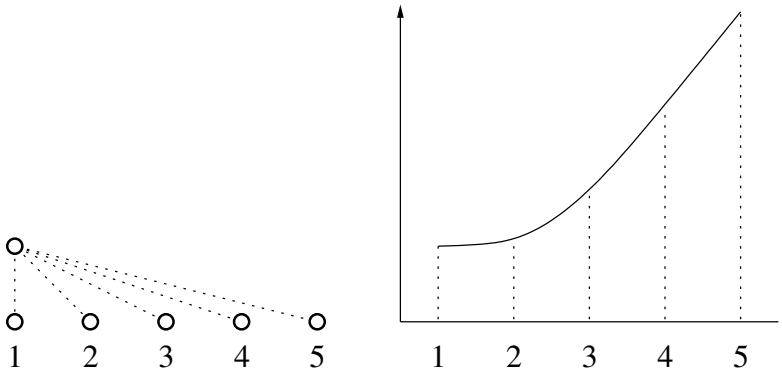


Figure 2.2: Distance of points on a line to a fixed point above the line. The distance is non-linearly dependent on the distance the points have on the line.

Unfortunately, no graphics cards exist offering the possibility to render four-dimensional scenes into three-dimensional buffers. Instead, one has to approximate each cone by a series of conic sections parallel to the axis of the cone. This approximation is illustrated in one dimension less in Figure 2.3. With the help of this approximation, it is then possible to render the three-dimensional discrete Voronoi diagram in two-dimensional layers.

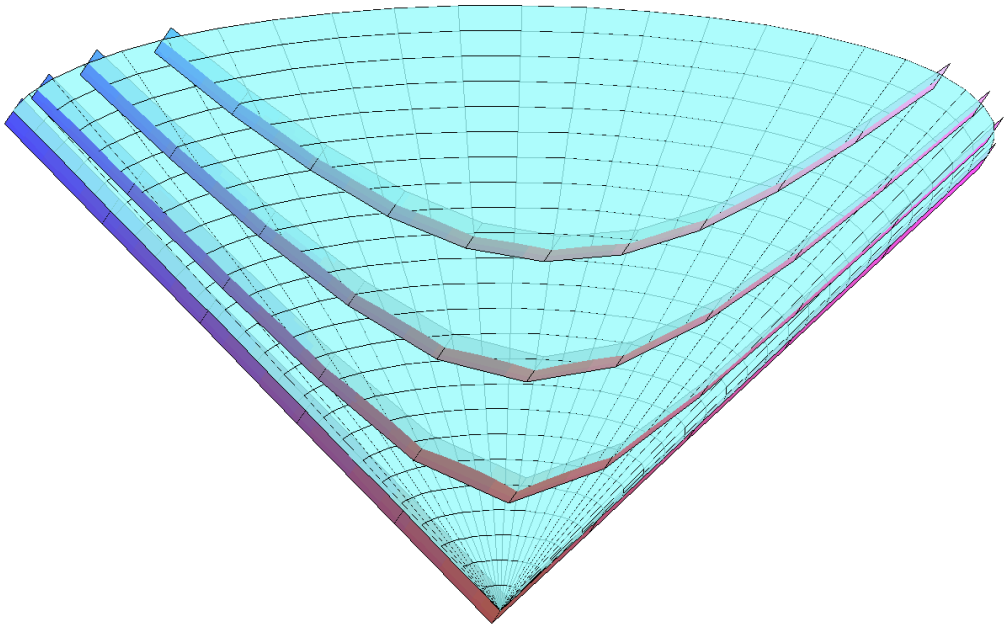


Figure 2.3: Approximation of one half of a two-dimensional cone by layers of conic sections.

First, all sample points have to be sorted into the given grid, i. e. each sample point is assigned to one grid cell and the cell is marked as filled. For each two-dimensional layer, the respective part of the discrete Voronoi diagram is calculated by rendering

a certain conic section for each filled grid cell. The appropriate conic section for a filled cell with midpoint $\mathbf{x} \in \mathbb{R}^3$ and orthogonal distance $t \in \mathbb{R}$ to the current layer is defined by

$$C_{\mathbf{x},t} := \left\{ (y_1, y_2, y_3) \in \mathbb{R}^3 \ : \ \sqrt{(y_1 - x_1)^2 + (y_2 - x_2)^2 + t^2} = y_3 \right\} .$$

The application of depth testing and orthogonal projection leads to an approximation of the discrete Voronoi diagram in the current layer, as illustrated in one dimension less in Figure 2.4. The process is repeated for all layers until the discrete Voronoi diagram is generated for the whole domain.

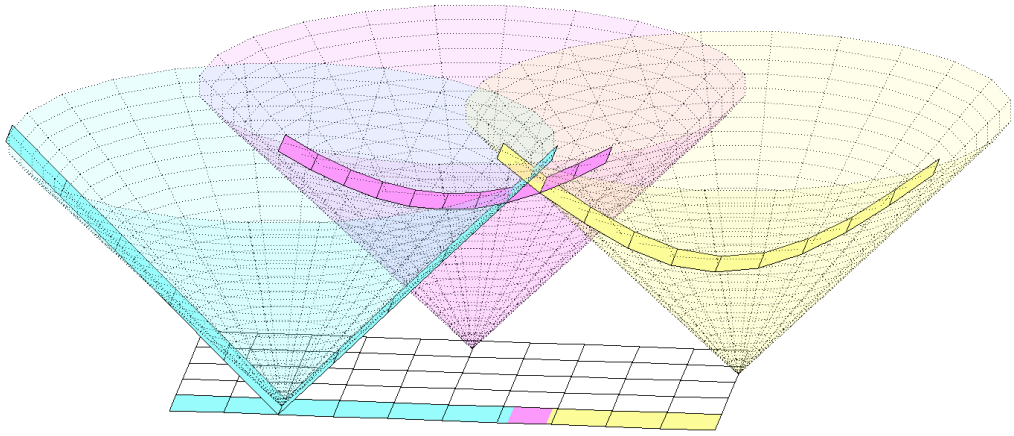
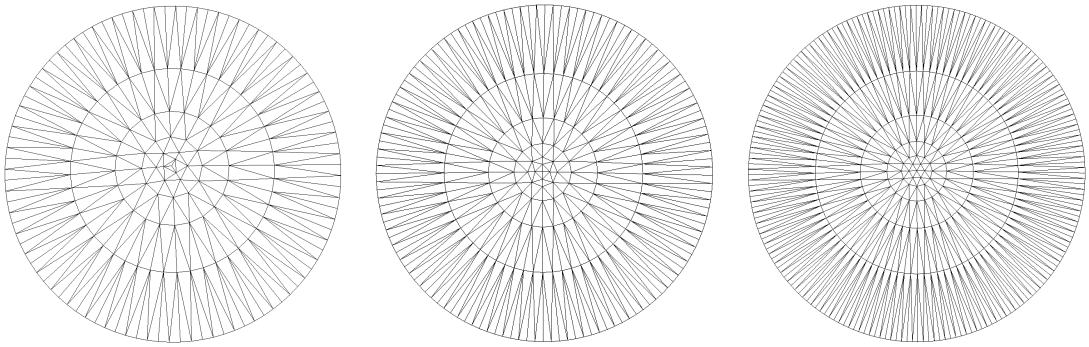


Figure 2.4: Generation of a one-dimensional layer of a two-dimensional discrete Voronoi diagram. For each sample point, one conic section is rendered above the layer with respect to the distance of the point to the layer. Using the depth buffer and orthogonal projection results in an approximation of the discrete Voronoi diagram in the current layer.

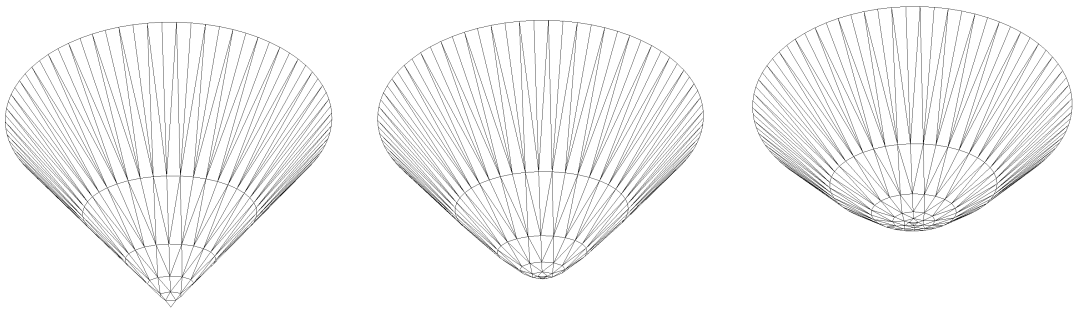
For the overall process, only a finite number of different conic sections, with $t = 0, 1, 2, \ldots, t_{\max}$ is needed. Here, $t_{\max}$ only depends on the size of the grid in x_3-direction. Hence, the set of needed conic sections can be precomputed and used throughout the whole process. One has to parameterize the conic sections and approximate them with triangles. The projections of different triangulations are shown in Figure 2.5.

The conic sections are parameterized in the canonical way with cylindrical coordinates. The triangulation is done in rings with growing radii. To assure a nearly constant triangle per diameter ratio, the triangles per ring are doubled just like the outer radius of each ring in cylindrical coordinates. The user can specify the number of triangles in the innermost ring and its radius. Given this information, the set of approximated conic sections can be built. An illustration of conic sections with different values of the parameter t is shown in Figure 2.6.



(a) (b) (c)

Figure 2.5: Projection of different triangulations for a conic section. The triangulation is obtained by utilizing the rotational symmetry of the conic section. Around one central point all other points are lying on circles with growing radii. For the pictures, different number of triangles were used in the first ring, starting from three in (a), four in (b) to six in (c). The number of triangles in the subsequent ring is always doubled.



$t = 0$ $t = 30$ $t = 100$

Figure 2.6: Illustration of triangulated conic sections with different values of the parameter t. All conic sections have the same inner radius $r = 6$ and three triangles in the innermost ring.

Let the grid cells be unit cubes. If the radius of the innermost ring is greater than one, the approximation error of the triangulation is bounded by the triangulation error in the first ring. For the presented approach, an inner radius of four was chosen with three initial triangles. This leads to an approximation error of less than one, i. e. the approximation error is below grid size.

After the precomputation of the set of conic sections, the main algorithm starts. For each slice of the volume and for each sample point, the correct conic section is taken from the look-up table and rendered to a texture using the depth buffer of the graphics card. Each sample point is assigned a certain color, which the respective conic section is rendered with. This process ends up in a stack of layered discrete Voronoi diagrams, representing a three-dimensional discrete Voronoi diagram with color coded Voronoi regions. An overview of this algorithm is given in Table 2.1.

```
BuildDVD()
{
  approximate set of conic sections
  for (each layer l)
  {
    clear texture and depth buffer
    for (each sample point x)
    {
      render conic section with t = |l − x3| at (x1, x2)
    }
    read back l from frame buffer
  }
}
```

Table 2.1: Algorithm for the generation of a three-dimensional discrete Voronoi diagram.

The natural neighbor candidates are afterwards extracted for each sample point. This is done by exploring the whole discrete Voronoi diagram for adjacent Voronoi regions. If such two regions are found, the points lying in the grid cells corresponding with the Voronoi regions are marked to be neighbor candidates for each other. Note that multiple points may fall into one grid cell. All pairs are neighbor candidates respectively. After finalizing this stage, also all sample points lying in one cell are marked as neighbor candidates for each other.

This whole process results in a list of neighbor candidates for each sample point, derived from the discrete Voronoi diagram information. To get the final approximation for the natural neighbors of each sample point, a set of criteria is applied to reject some of these candidates. This process is explained in detail in Section 2.3.

2.2 kd-tree-based Natural Neighbor Approximation

In Section 2.1 an approach for approximating Voronoi diagrams, which the natural neighbors are based on, is proposed. Instead of doing so, one can also think of directly approximating the natural neighborhood of each sample point.

To achieve this, one would first have to clarify the main properties of natural neighbors to find out how to approximate a set of neighbors that has very similar properties. The main property of the natural neighborhood of a sample point, which is also fundamental for the proposed surface extraction approach, is that the natural neighbors surround the point, i. e. the Voronoi cells of the natural neighbors of a sample point completely separate the point from the rest of the sample points.

This idea of completely surrounding the point by neighboring regions is adopted. Instead of surrounding a point with Voronoi regions, it should be surrounded by axis aligned cells, approximating the natural neighborhood. For this purpose, the sample points are stored in a kd-tree as described in Section 2.2.1. The process of finding neighbor candidates is based on a fast and efficient indexing scheme and is described in Section 2.2.2.

2.2.1 Unstructured Point-based Data Storage

Typically, unstructured data points are given by position in space coordinates and function value. To allow a fast approximation of the natural neighbors for the n unstructured sample points, they have to be stored in a three-dimensional kd-tree [Ben75, Ben90].

A three-dimensional kd-tree is a natural generalization of one-dimensional binary search trees [Knu98]. It is a hierarchical data structure that stores three-dimensional data points and is organized as an abstract tree with direct relation to the data point locations in space. Each node stores at most one data point and splits the domain of the data in two parts with respect to a two-dimensional cutting plane perpendicular to one of the coordinate axis. The coordinate axes are equal for nodes with the same depth in the tree. However, the used coordinate axes are alternating with growing depth. The whole three-dimensional kd-tree represents a partition of the domain in cuboids, called cells.

Because of memory-saving reasons and fast access via an indexing scheme, the points are not directly stored in the kd-tree, but in a vector of points v. Thereon the kd-tree is build recursively. The recursive function to build the tree is shown in pseudocode in Table 2.2. For every depth i and vector v, v is sorted in $x_{i \bmod 3}$-direction, where x_0, x_1, and x_2 denote the three dimensions. Afterwards v is split in two half-sized subvectors v_1 and v_2. The same procedure is recursively applied to the subvectors as long as they are not empty.

```
BuildTree(depth i, vector v)
{
  if (size(v) > 0)
  {
    sort v in x_{i mod 3}-direction
    if (size(v) mod 2 = 0)
    {
      insert median as split into tree
    }
    else
    {
      insert median element into tree
      remove median element from v
    }
    if (size(v) > 0)
    {
      bisect v into v_1 and v_2 with respect to median element
      BuildTree(i + 1, v_1)
      BuildTree(i + 1, v_2)
    }
  }
}
```

Table 2.2: Recursive function to build the kd-tree.

There are many ways of choosing the dimension to split in each of the kd-tree building steps, leading probably to more balanced space partitions. However, for applying the neighbor searching algorithm, described in the next sections, it is fundamental to use the cyclic algorithm shown in Table 2.2.

Each cell of the constructed kd-tree contains exactly one sample point. The height of the tree is $\lceil \log_2(n+1) \rceil$. In the worst case ($n = 2^j$), $j \in \mathbb{N}_+$, the number of nodes in the kd-tree is $2n - 1$. The whole tree is stored in a vector, where the root is in position 1 and the children of the node in position j are in positions $2j$ and $2j + 1$. This allows a fast traversal of the tree. If one interprets the positions of the nodes in the vector as binary numbers, the traversal gets even faster.

In the following, all integers indexed with d such as a_d or 100_d denote binary numbers. The operator $\oplus$ denotes the bitwise Boolean exclusive-or operator, such that $a_\mathrm{d} \oplus b_\mathrm{d}$ means a_d XOR b_d.

Definition 2.3 *The integer operators $\ll$ and $\gg$ denote the bit-shift operators, which are recursively defined by*

0. $a_\mathrm{d} \ll 0 = a_\mathrm{d} \quad$ *and* $\quad a_\mathrm{d} \gg 0 = a_\mathrm{d}$.

1. $a_\mathrm{d} \ll j = (a_\mathrm{d} \ll (j-1)) * 2 \quad$ *for all* $\quad j \in \mathbb{N}_+$.

2. $a_\mathrm{d} \gg j = (a_\mathrm{d} \gg (j-1)) \operatorname{div} 2 \quad$ *for all* $\quad j \in \mathbb{N}_+$.

The indices of the nodes in the vector can be interpreted as binary numbers. A two-dimensional example of this is shown in Figure 2.7. This point of view implies an indexing scheme for all nodes of the kd-tree.

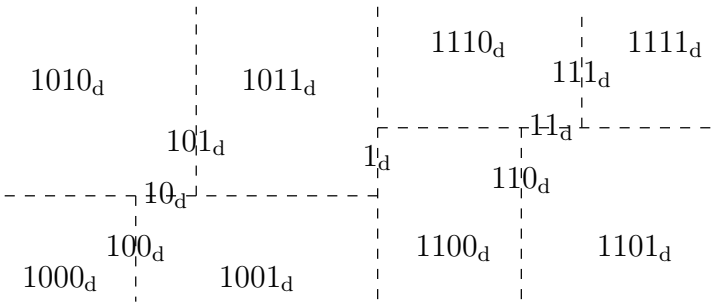


Figure 2.7: Two-dimensional kd-tree. The binary representation of the position in the tree is denoted for each cell and splitting line.

The parent of a node with index b_d is in position $b_\mathrm{d} \gg 1$ in the vector. The children of this node have the indices $b_\mathrm{d} \ll 1$ and $(b_\mathrm{d} \ll 1) \oplus 1_\mathrm{d}$.

Hence, the indexing scheme leads to fast position and relationship queries in the kd-tree. It is possible to navigate through the tree with fast binary operations. Moreover,

qualitative propositions about the locations of cells can be made. For instance, the cells in position 1111_{d} and 1000_{d} in Figure 2.7 lie in diagonally distant corners of the kd-tree. Many informations are implicitly saved in the indexing scheme and can be used for speeding up neighbor queries.

2.2.2 Neighbors Search

As shown in Section 2.2.1, the introduced indexing scheme gives insight into the kd-tree structure just by observing the nodes' binary positions. This is now used to find neighbor candidates for the sample points. The search for neighbor candidates is only done for sample points that lie inside a cell of the kd-tree.

Definition 2.4 *Let c be a cell of the kd-tree T. A cell or splitting plane of T is a subset of the surrounding neighborhood of c, iff it has at least one point in common with c. The set of indices of all surrounding neighbors of c is denoted by $N(c)$.*

A two-dimensional example of the surrounding neighborhood of the cell containing the point $\mathbf{x}$ is shown in Figure 2.8. It is clearly visible that the observed cell, marked with gray, is completely surrounded by the neighboring cells and splitting lines, colored in blue.

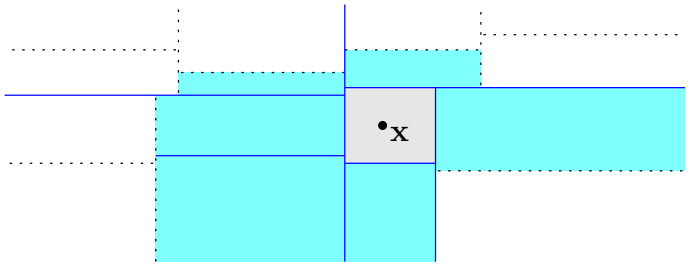


Figure 2.8: Surrounding neighborhood of the cell containing sample point $\mathbf{x}$. All light blue areas and blue lines belong to the neighborhood.

The way the nodes of the kd-tree are saved in the vector constitutes that the position of every node and leaf in the tree is clearly determined by the binary representation of its index in the vector. Thus, the cells and splitting planes can be identified with their index in the vector.

The surrounding neighbors of a cell can be divided into direct and indirect neighbors. This classification depends on the location of the neighbors, which is explained in the following sections in more detail.

2.2.3 Direct Neighbors

For finding surrounding neighbors of cells, the last three steps of the kd-tree building process, i. e. the last split in each of the three dimensions play an important role. All

direct neighbors are generated in these steps. More precisely the direct neighbors of a cell c are all neighboring cells and splitting planes that were generated in the last three steps of the kd-tree building process.

These neighbors can be easily found by observing the index of the cell and querying for certain split elements. For these considerations, let b_d be the position of the cell c in the vector.

The splitting planes with indices $b_\mathrm{d} \gg 1$, $b_\mathrm{d} \gg 2$, and $b_\mathrm{d} \gg 3$ are always direct neighbors of c, because they cover three faces of c. From this one can directly deduce

$$\{b_\mathrm{d} \oplus 1_\mathrm{d}, b_\mathrm{d} \oplus 10_\mathrm{d}, b_\mathrm{d} \oplus 100_\mathrm{d}\} \subset N(c)$$

as direct neighbors.

Figure 2.9 shows a three-dimensional example of a neighborhood configuration. To obtain the remaining direct neighbors, the splitting planes that are beyond the already inserted planes have to be checked.

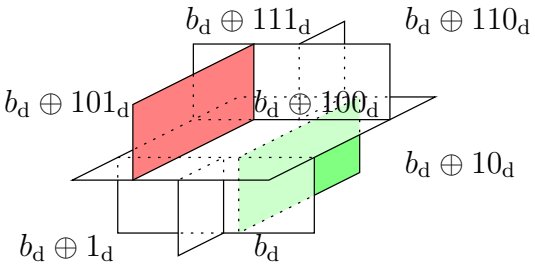


Figure 2.9: Direct neighbors of the cell with index b_d. The green splitting plane touches the cell with index b_d, in contrast to the red plane.

For example, the light red splitting plane in Figure 2.9 between $b_\mathrm{d} \oplus 101_\mathrm{d}$ and $b_\mathrm{d} \oplus 100_\mathrm{d}$ corresponds to index $(b_\mathrm{d} \gg 1) \oplus 10_\mathrm{d}$. It has no common point with c and, hence, does not belong to $N(c)$. Therefore $b_\mathrm{d} \oplus 101_\mathrm{d}$ also can not belong to $N(c)$. The other case occurs for $(b_\mathrm{d} \gg 1) \oplus 1_\mathrm{d}$, the light green plane in Figure 2.9. This plane belongs to $N(c)$ and we deduce $b_\mathrm{d} \oplus 11_\mathrm{d} \in N(c)$.

With some more considerations along the same lines one gets the following conditions to obtain the remaining direct neighbors:

$$\begin{aligned}
(b_\mathrm{d} \gg 1) \oplus 1_\mathrm{d} \in N(c) \quad &\Rightarrow \quad b_\mathrm{d} \oplus 11_\mathrm{d} \in N(c) \\
(b_\mathrm{d} \gg 1) \oplus 10_\mathrm{d} \in N(c) \quad &\Rightarrow \quad b_\mathrm{d} \oplus 101_\mathrm{d} \in N(c) \\
(b_\mathrm{d} \gg 2) \oplus 1_\mathrm{d} \in N(c) \quad &\Rightarrow \quad b_\mathrm{d} \oplus 110_\mathrm{d} \in N(c) \\
\{(b_\mathrm{d} \gg 2) \oplus 1_\mathrm{d}, (b_\mathrm{d} \gg 1) \oplus 11_\mathrm{d}\} \subset N(c) \quad &\Rightarrow \quad b_\mathrm{d} \oplus 111_\mathrm{d} \in N(c)
\end{aligned}$$

In fact, all direct neighbors of c can be found by applying binary operations on b_d and at most four comparisons of splitting values from the kd-tree nodes. The obtained neighborhood covers half of the faces of the cell completely.

2.2.4 Indirect Neighbors

In a second step, the neighbors for the three other faces of each cell c, called indirect neighbors, have to be obtained. In contrast to the direct neighbors, the indirect neighbors result from arbitrary steps of the kd-tree generation and the maximum number of indirect neighbors is not constant.

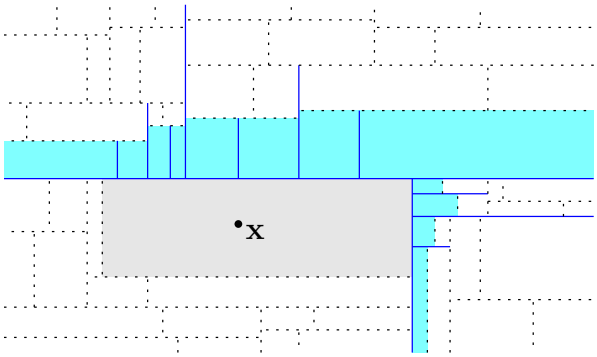


Figure 2.10: Indirect neighbors of $\mathbf{x}$ (blue).

The number of cells of a kd-tree for n sample points is $O(n)$. This leads to an upper bound for the number of indirect neighbors. In two-dimensional case, which one can see in Figure 2.10, the maximum number of indirect neighbors is $O\left(\sqrt{n}\right)$, whereas it is in the three-dimensional case $O\left(\sqrt[3]{n^2}\right)$.

The given upper bounds for indirect neighbors can, in fact, only occur for a small number of cells in a constructed worst-case scenario. On average, the cell sizes of neighbored cells in a kd-tree differ not more than one order of magnitude. Consequently, the average case for a cell in three dimensions has about nine neighbors for each face, leading to 27 indirect neighbors per cell.

To compute the indirect neighbors, the three splitting planes that cover the remaining faces of the cell c have to be found. Obviously, there is exactly one splitting plane for each dimension. This splitting plane is always opposite to an already found direct neighbor splitting plane, i. e. both splitting planes are perpendicular to the same direction. This directly implies that if the direct splitting plane has l digits in its binary index representation the indirect splitting plane has $l - 3j$ digits for a $j \in \mathbb{N}_+$.

Furthermore the splits of the planes go in different directions. That means, if the observed cell lies above the direct splitting plane, it lies below the indirect splitting

plane and vice versa. Expressed in the binary indexing scheme this means that if the direct splitting plane has a 0_d as the last bit, the indirect splitting plane has to have a 1_d there.

> Let b_d be again the position of c in the vector. The three splitting planes that cover the remaining faces of the cell c can be found by analyzing the sequence of digits in b_d. One has to seek b_d from back to front for bit changes in the binary sequence of each dimension. If j is the digit, where the first bit change in the dimension appears, then $b_\mathrm{d} \gg j$ is the searched splitting plane. If the sequence corresponding to the dimension is constant, then there are no indirect neighbors for this dimension, due to the fact that the cell lies at the outer border of the whole domain.

Example 2.5 Let $b_\mathrm{d} = 1001000011001_\mathrm{d}$. We first search a skip in the x_0-sequence, illustrated at the top of Figure 2.11. In this sequence only the digit 0 appears. That means c lies in negative x_0-direction of every splitting plane. So c has no bounding plane in negative x_0-direction and also no indirect neighbors in this dimension.

$$\begin{aligned} &1001000011001 - x_0 \\ &1001000011001 - x_1 \\ &1001000011001 - x_2 \end{aligned}$$

Figure 2.11: Example of search for bit changes in each dimension to determine the indices of the splitting planes (blue).

The second search, in the x_1-sequence, delivers a change from 0 to 1 in the bit at the fifth position counted from the back of b_d. So the splitting plane which bounds c in negative x_1-direction is 10010000_d. Similarly one gets the second and last plane of the boundary of c. In positive x_2-direction it is 100100_d. □

In the first phase of the search for indirect neighbors at most three splitting planes have been calculated that cover the remaining faces of the cell. Obviously, all other indirect neighbors have to align to one of the planes. For each of those splitting planes p, one has to search for all cells and planes that are at the opposite side of p with respect to c and belong to the surrounding neighborhood of c.

All of these searched cells and planes have a deeper depth in the kd-tree than p. Another property, that follows immediately is, that all searched planes are never parallel to p, i. e. they are no splitting planes in the same dimension as p.

With these properties a recursive search is done, which starts in the subtree of p that does not contain c. The recursive function that performs this search is described in Table 2.3.

This recursion gradually checks splitting planes for being indirect neighbors. It ends in cells of the kd-tree which also belong to the surrounding neighborhood. After

```
SubtreeSearch(depth i, depth p, node k, cell c)
{
  if (k is a leaf)
  {
    insert k into N(c)
  }
  else
  {
    if ((i − p) mod 3 = 0)
    {
      SubtreeSearch(i + 1, p,k → child next to c, c)
    }
    else
    {
      if (plane has a common point with c)
      {
        insert k into N(c)
        SubtreeSearch(i + 1, p,k → left child, c)
        SubtreeSearch(i + 1, p,k → right child, c)
      }
      else
      {
        SubtreeSearch(i + 1, p,k → child next to c, c)
      }
    }
  }
}
```

Table 2.3: Recursive function to search for the indirect neighbors of c lying on the opposite side of the splitting plane p. Here, i is the depth of the actual observed node k and p is the depth of the splitting plane separating the subtree from cell c.

executing this recursion for all found dividing planes all indirect neighbors of c are found.

Example 2.6 For the observed cell from Example 2.5, $p_\mathrm{d} = 10010000_\mathrm{d}$ was a found splitting plane for $b_\mathrm{d} = 1001000011001_\mathrm{d}$. The start of the recursive search would be 100100000_d. Assuming that this plane lies in positive x_1-direction of c, i. e. has no common point with c, one proceeds with 1001000000_d. If this plane has a common point with c, it has to be inserted into $N(c)$ and the binary search continues with 10010000000_d and 10010000001_d. Because these planes are three steps deeper than p_d in the kd-tree, they are dividing in the same dimension as p_d did. So the binary search has to proceed with 100100000001_d and 100100000011_d. The recursion stops in the next step by obtaining cells belonging to $N(c)$. □

As one can see from the considerations above, only the test if a plane has a common point with a cell c uses geometric queries, i. e. the dimensions of c and the pivot value of the splitting. All other operations for searching the kd-tree only use the binary representation of c and fast binary operations. Thus, the search for surrounding neighbors has a performance that is similar to nearest neighbor search in kd-trees.

2.3 Angle Criterion

In Sections 2.1 and 2.2, a set of neighbor candidates was obtained for each sample point. However, both of the approaches only assure a dense covering of each sample point's surrounding with neighbor candidates. A second fundamental property of natural neighbors is the distribution around the point, proportional to the local point distribution. This property is hardly ever assured in the presented approximation approaches. This can lead to very undesirable situations, when applying linear interpolation between neighboring sample points, as shown in Figure 2.12.

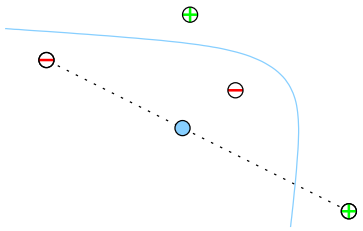


Figure 2.12: Example where linear interpolation between two-dimensional neighbor candidates fails. Sample points with function values below the isovalue are marked red, while points with values above the isovalue are marked green. The interpolated isopoint, marked light blue, is farther away from the light blue isocontour than all sample points.

Such bad situations can be avoided by restricting the minimum angle between two neighbor points of each sample point, respectively. This would prevent neighbors from lying behind each other and is an adaptation of the angle criterion method Linsen and Prautzsch [LP01, LP03] used for point-based surface representations.

The minimum angle that is used for the angle criterion is motivated by the regular grid case, shown in Figure 2.13. For a sample point on a regular grid, one would definitely want the points directly connected by the grid to be neighbors. These neighbors are colored blue in Figure 2.13. The second category of points that should be neighbors are the sample points that lie diagonal to the midpoint with respect to the grid, colored in green.

This choice is consistent with the optimal sphere packing problem in three dimensions [Hal05]. The minimum angle between these chosen neighbors in the regular case is

$$\alpha = \cos^{-1}\left(\sqrt{\frac{1}{3}}\right) \approx 54.736^\circ \ .$$

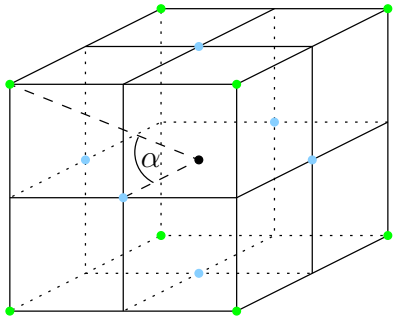


Figure 2.13: Minimal angle α between neighbors derived from the regular case. In this case it is reasonable to define all direct connected points, colored blue, and all diagonal points, colored green, as neighbors.

The above described property has to be ensured for the final neighbors of each sample point. To achieve this, all points in the set of neighbor candidates have to be checked in terms of this requirement. For each point $\mathbf{x}$, the neighbor candidates are first sorted according to their quadratic Euclidean distance from $\mathbf{x}$. Afterwards the list of remaining neighbor candidates is traversed with increasing distance.

For each neighbor candidate, the cosine of the angle to the previously checked neighbors is tested for being smaller than $\sqrt{\frac{1}{3}}$. If the neighbor candidate fails this minimum angle test it is removed from the list of neighbors. This procedure is illustrated in Figure 2.14. A detailed description of the algorithm for finding the final neighbors of each point $\mathbf{x}$ with the help of a list of neighbor candidates v is given in Table 2.4.

With applying the criterion described above to the lists of neighbor candidates resulting from the algorithms of Sections 2.1 or 2.2, a good approximation of natural

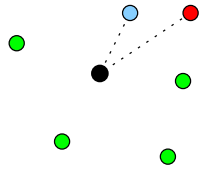


Figure 2.14: Two-dimensional angle criterion test. All neighbors of the black point are tested one after the other according to their angle to the closer neighbors. The angle between the red and the blue colored neighbor is too small. The red neighbor is discarded and the other points are confirmed as appropriate neighbors.

FinalNeighborSearch(point $\mathbf{x}$, vector of neighbors v)
{
  sort v according to distance to $\mathbf{x}$
  for(all points $\mathbf{y}$ in v)
  {
    for(all previous points $\mathbf{z}$ in v closer than $\mathbf{y}$)
    {
      if$\left(\frac{\langle \mathbf{y}-\mathbf{x}, \mathbf{z}-\mathbf{x} \rangle}{\|\mathbf{y}-\mathbf{x}\| \|\mathbf{z}-\mathbf{x}\|} > \sqrt{\frac{1}{3}}\right)$
      {
        remove $\mathbf{y}$ from v
      }
    }
  }
}

Table 2.4: Function to find the final surrounding neighbors for each point $\mathbf{x}$ by applying the minimum angle criterion. The neighbor candidates are given in a vector v. After execution of the function only the final neighbors remain in the vector v.

neighbors is found for each sample point. The approximation takes into account the proportional distribution of the neighbors with respect to the local point distribution as well as the surrounding character of natural neighbors.

2.4 Isopoint Calculation

As a result from the prior steps, a neighborhood of points is generated for each sample point $\mathbf{x}$ approximating the natural neighborhood. Following the idea of several well known marching algorithms the isopoints are linearly interpolated between neighboring sample points on opposite sides of the isosurface with respect to the function values f and the isovalue f_{iso}.

For each sample point $\mathbf{x}$, the whole surrounding neighborhood is searched for appropriate interpolation candidates, i. e. for each neighbor $\mathbf{y}$ the term

$$(f(\mathbf{x}) - f_{\text{iso}})\,(f(\mathbf{y}) - f_{\text{iso}})$$

is checked for being negative. If this is fulfilled a new isopoint $\mathbf{z}$ is linearly interpolated between $\mathbf{x}$ and $\mathbf{y}$ with respect to the function values and the isovalue, i. e.

$$\mathbf{z} = \left|\frac{f_{\text{iso}} - f(\mathbf{y})}{f(\mathbf{x}) - f(\mathbf{y})}\right| \cdot \mathbf{x} + \left|\frac{f(\mathbf{x}) - f_{\text{iso}}}{f(\mathbf{x}) - f(\mathbf{y})}\right| \cdot \mathbf{y}\ .$$

The interpolated isopoints exhibit no surface normals, which are needed for rendering the isosurface. Instead, they can be provided with an outside vector giving at least the orientation of the surface. With this it is easy to approximate consistent surface normals. The assigned outside vector is calculated as

$$\mathbf{v} = (f(\mathbf{x}) - f(\mathbf{y}))\,(\mathbf{x} - \mathbf{y})\ .$$

Because of the angle criterion, the neighborhood creation is not symmetric, i. e. it is possible that $\mathbf{x}$ is not neighbor of all sample points in its surrounding neighborhood. Thus, symmetry can not be exploited and some isopoints may be computed twice. Such duplicates are removed in a final step. The isopoints are sorted lexicographically regarding the coordinate axes. Duplicates are afterwards direct neighbors in the sorted list and are removed quickly.

2.5 Results and Discussion

In this section the achieved results from the algorithms above are presented. The different approaches are applied to several synthetic and real data sets and they are discussed in matter of performance and quality. All measurements were taken on an 2.66GHz Intel Xeon processor with an Nvidia Quadro FX 4500 graphics card.

First both approaches for approximating natural neighbors, the discrete Voronoi diagram generation and the kd-tree-based natural neighborhood approximation, are

compared in terms of performance. The time for the discrete Voronoi diagram generation is obviously dependent on the number of grid cells filled with sample points. Thus, it is also implicitly dependent from the chosen grid size, especially if there exist cells which include several sample points.

A comparison of the computation times for different grid sizes for just the discrete Voronoi diagram generation can be seen in Table 2.5. The GPU-based algorithm was applied to a synthetic data set, with sample points at randomly distributed positions in a unit cube. For different resolutions of the underlying grid, the number of cells filled with at least one sample point is given just as the overall computation time and the computation time per one million voxels. As a consequence of the depth-testing capabilities of the GPU, the computation time for the discrete Voronoi diagram generation increases sublinearly with increasing number of voxels.

grid size	**filled cells**	**comp. time**	**comp. time per 1M voxels**
32^3	8,571	3 sec	91 sec
64^3	9,828	9 sec	34 sec
128^3	9,976	40 sec	19 sec
256^3	9,996	139 sec	8 sec

Table 2.5: Computation times for the discrete Voronoi diagram generation for a data set with 10,000 randomly placed sample points. For each grid size, the number of cells with sample points, the overall computation time, and the computation time per million grid voxels is given. The radius of the innermost ring of each conic section was $r = 2$ with 4 triangles.

The computation times for the discrete Voronoi diagram generation in comparison to the number of filled grid cells are shown in Table 2.6. On a grid with 256^3 cells a number of random grid cells have been marked as filled and the discrete Voronoi diagram was computed. Obviously, the computation time is linearly dependent on the number of filled grid cells, but increases sublinearly with increasing ratio of filled cells. This is due to the fact, that most of the triangles of the conic sections are directly omitted by the depth test during the rendering.

filled cells	1,250	2,500	5,000	10,000	20,000
comp. time	21 sec	39 sec	81 sec	139 sec	253 sec

Table 2.6: Computation times for the discrete Voronoi diagram generation in comparison to the number of filled grid cells of a grid with 256^3 cells. The radius of the innermost ring of each conic section was $r = 2$ with 4 triangles.

The practicability of the approach is shown on astrophysical data sets, simulating black hole encounters of white dwarfs [RRHD08]. Slices of the computed discrete Voronoi diagram of the volume data set consisting of 500,000 unstructured sample points are shown in Figure 2.15. The discrete Voronoi diagram was computed on a

$256 \times 256 \times 475$ grid. A set of six x-y-planes is extracted for different z-values. The Voronoi regions are colored respectively to allow a good perception of the three-dimensional shape of the discrete Voronoi diagram. One also gets a good impression of the highly varying point density of these data sets.

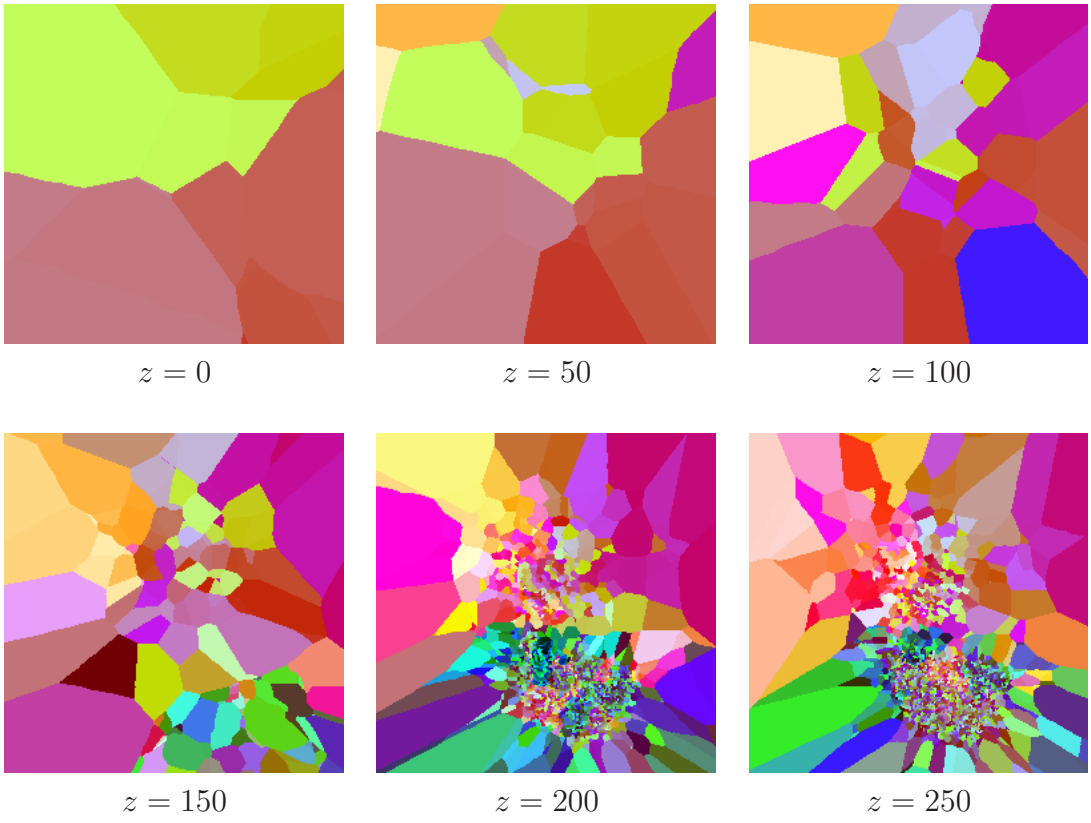


Figure 2.15: Slices from the discrete Voronoi diagram generated for the white dwarf data set with 500k unstructured sample points discretized on a $256 \times 256 \times 475$ grid. The radius of the innermost ring of each conic section was $r = 2$ with 4 triangles. The pixels of the slices are colored with random colors with respect to the Voronoi regions.

After having assigned the sample points to the grid, the number of sample points per grid cell can be very high, as shown in Table 2.7. This means that many points have to be discarded by the angle criterion leading to additional computation time in this step. Nevertheless, it is obvious that most of the computation time is spent on generating the discrete Voronoi diagram, especially at high resolutions of the underlying grid.

Since the goal of the discrete Voronoi diagram generation is not achieving a perfect approximation of the Voronoi diagram, but just a good guess for the natural neighbors, it was sufficient for all computations and tests, to choose the radius of the innermost ring of each conic section as $r = 2$ with 4 initial triangles.

For the second approach, the kd-tree-based natural neighbor approximation, the same two test data sets were used. The overall natural neighbor approximation is

data set	white dwarf	
sample points	500k	4,000k
grid dimensions	$128 \times 128 \times 238$	$128 \times 128 \times 269$
filled cells	63,386	9,471
average	8 samp./cell	422 samp./cell
maximum	2,305 samp./cell	54,405 samp./cell
dVd generation	580 sec	100 sec
neighb. cand. calc.	5 sec	49 sec
final neighb. calc.	1 sec	4 sec
dVd neighbors	24	441
final neighbors	6	7

Table 2.7: Comparison of the whole discrete Voronoi diagram (dVd) calculation pipeline for two different data sets of the white dwarf simulation. For each data set, the resolution of the grid, the number of filled grid cells after insertion of the sample points, and the average and maximum number of sample points per filled grid cell are given. The computation times for the different steps of the pipeline are stated as well as the number of neighbors before and after applying the angle criterion.

data set	white dwarf	
sample points	500k	4,000k
***k*d-tree generation**	0.8 sec	8.9 sec
neighb. cand. calc.	0.7 sec	9.2 sec
final neighb. calc.	0.2 sec	1.0 sec
***k*d-tree neighbors**	11	11
final neighbors	5	5

Table 2.8: Results for the complete kd-tree-based natural neighbor approximation pipeline applied to two different data sets from the white dwarf simulation. For each data set, the computation times for the kd-tree generation, the neighbor candidates calculation, and for the application of the angle criterion are given as well as the average number of neighbor candidates and final neighbors.

significantly faster than using discrete Voronoi diagrams. The computation times for the whole computation pipeline are presented in Table 2.8. It is obvious that the kd-tree-based approach is much more flexible and robust against the very high varying sample point density of the astrophysical data sets. This is mainly because of the very high adaptability of the kd-tree data structure compared to the regular grid of the discrete Voronoi diagram.

From the previous considerations it is clearly visible that the natural neighbor approximation based on discrete Voronoi diagrams is only nearly competitive with the kd-tree-based approximation for very low resolutions of the underlying grid. That is why the question about the accuracy of both approaches arises. Here accuracy does not mean how many of the real natural neighbors are found by the approximations, but how far the calculated isopoints lie away from the real isosurface.

method	dVd-based	kd-tree-based
maximum deviation	0.02393	0.03490
relative deviation	0.00181	0.00199
computation time	361 sec	20 sec

Table 2.9: Comparison of discrete Voronoi diagram (dVd) and kd-tree-based natural neighbor approximation in terms of accuracy. For this purpose, a synthetic data set with four million sample points, representing a radial function was used. The sample points were randomly placed in a cubic volume. For the approach based on discrete Voronoi diagrams, a grid of size 64^3 was used. The deviations were measured as averages over ten test runs respectively.

To compare the deviations of the isopoints, both approaches were applied to a synthetic data set with four million data points at randomly sampled positions, representing a radial scalar field. Isopoints for the same isovalue were extracted using both approaches with the same angle criterion. Afterwards the average and maximum deviation of the respective isopoints were calculated. The resulting deviations are shown in Table 2.9. One can conclude that the high effort in computation time is not worth the small gain in precision when using the natural neighbor approximation based on discrete Voronoi diagrams instead of the kd-tree-based approach.

In comparison, the extraction of isopoints by the standard Marching Cubes algorithm [LC87] from a regular data set with 160^3 grid points representing the same scalar field leads to a maximum deviation of 0.0027. Hence, the isosurface extraction from regular data is about one order of magnitude more precise than the direct isosurface extraction from unstructured point-based data.

The reduction of the grid size to 32^3 for the isopoint computation, based on discrete Voronoi diagrams, would reduce the computation time to 112 seconds. Further reduction of the grid size would not reduce the computation times further, since the number of neighbor candidates grows enormously. This behavior is illustrated in Figure 2.16. The overall computation time is plotted against the side length of the underlying grid using a logarithmic scale. The locally minimal computation times

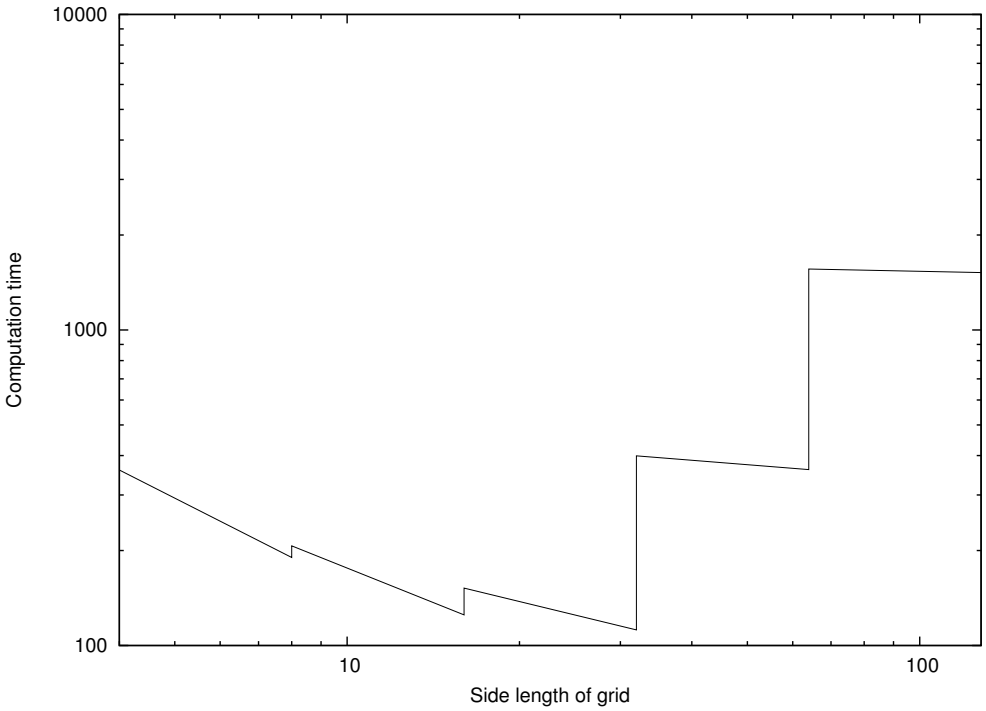


Figure 2.16: Graph of the overall computation time for the natural neighbor approximation based on discrete Voronoi diagrams subject to the size of the underlying grid for a synthetic data set with four million sample points. The computation time is given in seconds, the side length of the cubic grid is given in a logarithmic scale.

are achieved by using grid sizes which are powers of two, since the computations on the GPU are optimized for this type of textures. Using a slightly smaller grid size does not decrease the computation time for generating the discrete Voronoi diagram. On the other hand, the number of points per cell increases as well as the computation time for applying the angle criterion, leading to a higher overall computation time.

Figure 2.17: Splat-based ray tracing of the isosurface extracted with kd-tree-based isosurface extraction from the radial data set with 16 million sample points.

A rendering of the extracted spherical isosurface can be seen in Figure 2.17. Here the kd-tree-based isosurface extraction method was applied to a data set with 16 million uniform randomly distributed sample points representing a radial function. The isosurface was rendered using the splat-based ray tracing technique explained in Section 4.2.

Finally the direct isosurface extraction method based on kd-trees was applied to resampled unstructured point-based data sets. The data sets were obtained from regular data sets by resampling them at uniform randomly distributed sample positions. The first data set with eight million sample points was generated from the regular engine data set of size $256 \times 256 \times 128$. The whole process of building the kd-tree, finding neighbor candidates, applying the angle criterion, and extracting the 1,300k isopoints lasted only 96 seconds, whereas the isopoint extraction step took half of the overall time. A visualization of the extracted isosurface using the

Figure 2.18: Image-space point-cloud rendering of the isosurface generated with the kd-tree-based isosurface extraction approach from the engine data set with eight million sample points. (Data set courtesy of General Electric.)

image-space point-cloud rendering described in Section 4.3 is shown in Figure 2.18.

The second data set with 16 million sample points was generated from the regular Boston teapot data set of size $256 \times 256 \times 178$. The execution of the whole kd-tree-based isosurface extraction pipeline generated 243,000 isopoints and took 202 seconds of computation time. A rendering of the extracted isopoints is shown in Figure 2.19. Here only the lit point cloud was rendered to show the good extraction of the outer and inner border of the teapot. Also smooth transitions, especially on the spout and the knob, can be observed easily.

Comparing both presented approaches, it turns out that the latter is more flexible especially for the irregular point-based data sets often produced by many applications. The small gain in accuracy for the approach based on discrete Voronoi diagrams is not worth the loss in computation time compared to the kd-tree-based approach in many sample data sets. Reducing the used grid size would also not overcome this problem, since the produced neighbor candidates would grow enormously and the saved computation time would have to be reinvested in the following step of finding the final neighbors.

Finally, we compare our results to those obtained by Co et al. [CJ05]. The authors propose the only method that can be directly compared to our approach, as it is also able to directly operate on unstructured point-based data. However, our presented kd-tree-based method achieves a significantly better performance in terms of computation time. In their paper the authors state an isosurface extraction time of

Figure 2.19: Rendering of the lit isopoints generated by the kd-tree-based isosurface extraction method from the Boston teapot data set with 16 million sample points. (Data set courtesy of Terarecon Inc., MERL, and Brigham and Women's Hospital.)

470 seconds for a very small data set consisting of only 350,000 unstructured sample points, although the computations have been done on a cluster of 11 computers. The authors did not apply their method to larger data sets. Even when considering the development of computer hardware, we achieved a speed-up of several orders of magnitude with no visual shortfall in terms of quality.

2.6 Surface Extraction from Multi-variate Data

The presented isosurface extraction techniques are mainly designed and best applicable for visualization of scalar volume data. However, they can be also used to extract surfaces from unstructured multi-dimensional or multi-variate data. Instead of representing a three-dimensional scalar function, the unstructured data set M may consist of a set of data points in $\mathbb{R}^m$ or may represent a function

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^m \quad \text{with} \quad n, m \in \mathbb{N} \, .$$

One approach for visualizing this type of data is clustering with respect to different properties, i. e. mapping the points of data set M into a finite set of cluster classes C. We will now show how such clusters can be visualized, utilizing our direct isosurface extraction approaches. Therefore, one has to distinguish between volumetric multi-variate data ($n = 3$, $m > 1$) and multi-dimensional data with no spatial reference.

In the first case of volumetric multi-variate data, the data domain is $\mathbb{R}^3$. The clustering can take into account either only the data values in $\mathbb{R}^m$ or also the positions of the data points in $\mathbb{R}^3$. Anyway, it will result in a partition of the volumetric data set into different clusters. The proposed isosurface extraction approaches can be utilized to visualize the three-dimensional shape of each cluster in different ways.

For visualizing the shape of one cluster, a segmenting surface that separates the cluster points from the other data points could be extracted. Therefore, the cluster membership information can be directly encoded into a binary field at the data points. Each data point belonging to the cluster is assigned function value 1, while all other data points get function value 0. Applying direct isosurface extraction with isovalue 0.5 to the binary data set results in a surface segmenting the data domain with respect to the cluster membership property.

We applied this method to the data set of 2008 IEEE Visualization Design Contest [LLR09, RLL08, WN08]. Due to the large size, the provided regular data set was resampled. This was done at non-equidistant positions to avoid resampling artifacts, resulting in an unstructured multi-variate data set. The data set represents a time slice of the simulation of a front ionizing a gas. A hierarchical density-based clustering [LLRR08] was applied to extract clusters of interest. A splat-based ray tracing of the surface separating the cluster of ionized gas from the neutral gas is shown in Figure 2.20.

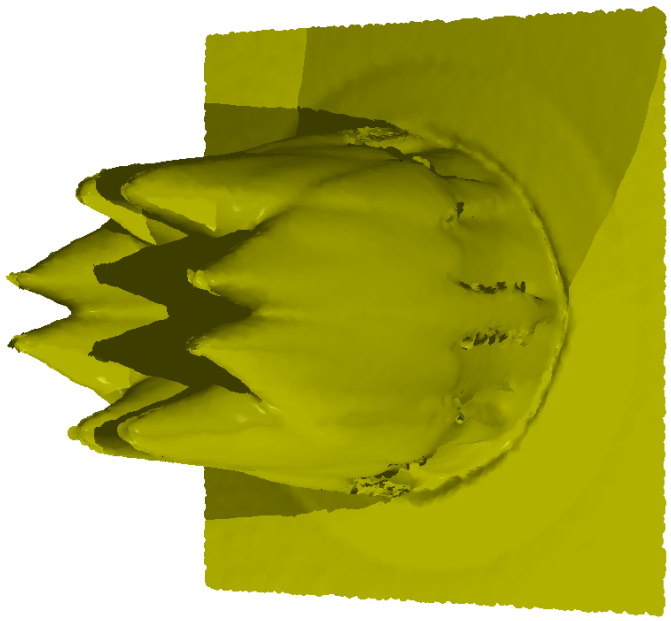
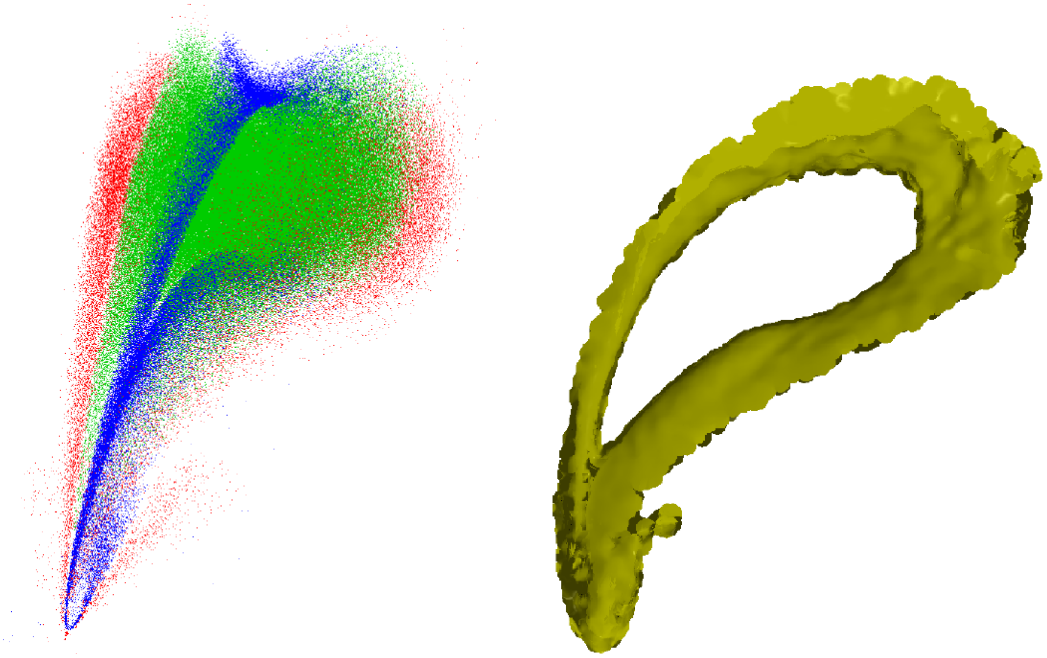


Figure 2.20: Splat-based ray tracing of the surface segmenting the cluster of ionized gas from the environment. The unstructured data set, subject of the 2008 IEEE Visualization Design Contest, simulates the propagation of an ionization front through a gas. Note that the boundary of the segmenting surface results from the boundary of the data domain.

This cluster visualization can be further improved if additional knowledge about the data set is available. For example for clusters obtained from smoothed particle hydrodynamics data, one can utilize the smoothing kernel to obtain a smoothed membership function for the data points. This will typically result in a smoother segmenting surface when extracting the isosurface to the isovalue 0.5. An example showing the color-coded clusters and a splat-based ray-tracing of a segmenting surface for a smoothed particle hydrodynamics data set with 500,000 particles is shown in Figure 2.21. The same effect would be achievable if the cluster algorithm provides a probability for the cluster membership of each data point.



(a) (b)

Figure 2.21: Cluster visualization for a smoothed particle hydrodynamics data set with 500,000 particles. The particles are clustered in three groups, which are rendered in different colors (red, green, blue) in (a). A splat-based ray-tracing of the surface separating the blue cluster from the other particles is shown in (b).

A second technique for visualizing clusters, obtained from unstructured volume data, would be to use a discrete Voronoi diagram, as described in Section 2.1, to generate a distance field from the cluster points [RL09]. The distance field can be computed without any additional effort during the generation of the discrete Voronoi diagram by recording the values of the depth buffer. A point rendering in feature space of two clusters from the 2008 IEEE Visualization Design Contest data set is shown in Figure 2.23 (a). Applying standard techniques for isosurface extraction to this distance field with an isovalue greater than the largest distance in the minimum spanning tree of the cluster points, results in a surface isodistant to the cluster points, cf. Figure 2.22 (a). A rendering of surfaces isodistant to the cluster points of

Figure 2.23 (a) are shown in Figure 2.23 (b).

A second cluster visualization, obtainable from the distance field to the cluster points is a cluster hull, enclosing the cluster by connecting cluster points. In contrast to the convex hull of the cluster points, the cluster hull can be non-convex. A surface isodistant to the point cluster consists of surface patches that are induced by the nearest cluster point. Hence, the neighborhood relation of the surface patches also creates a neighborhood information on the points. This neighborhood information can directly be obtained from the discrete Voronoi diagram by investigating natural neighborhoods in the isosurface region. When three neighborhoods come together, the respective points of the point cluster can be connected with a triangle. Generating all those triangles leads to a hull in form of a closed surface, as illustrated in Figure 2.22 (b). The resulting cluster hull for the cluster points from Figure 2.23 (a) is shown in Figure 2.23 (c). Here, the advantage compared to a convex-hull visualization becomes obvious, as the concave parts of the cluster's shape are well-preserved. This approach of generating a hull for points is similar to the algorithm of three-dimensional alpha shapes [EM94]. The choice of the isovalue for the hull isodistant to the cluster points correlates with choosing the parameter α of the alpha-shapes algorithm.

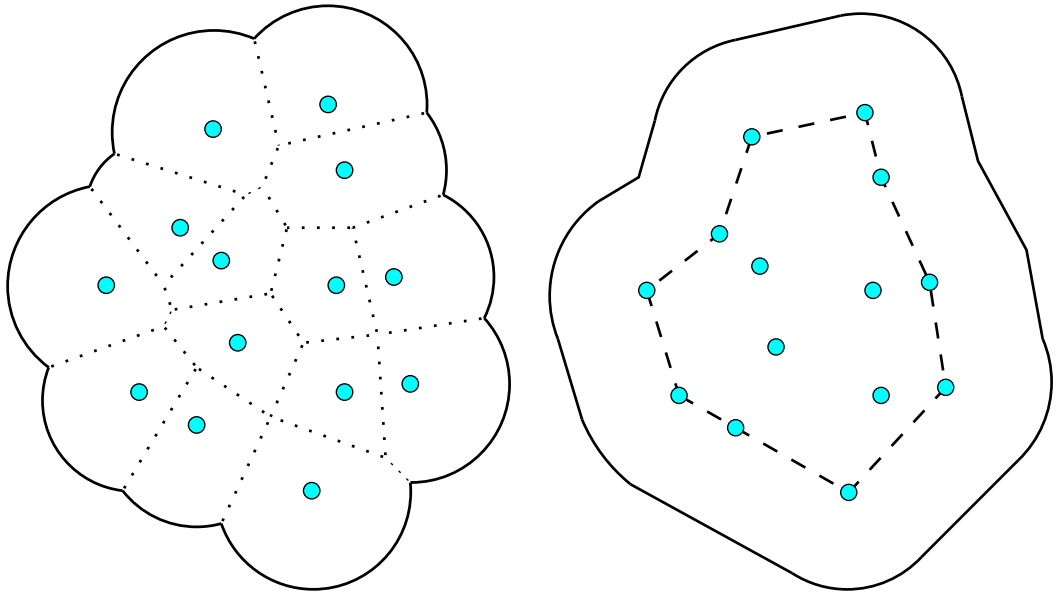


Figure 2.22: Hull generation for a point cluster: (a) Extracting an isosurface from the distance field to the point cluster. Voronoi regions on the isosurface induce neighborhoods. (b) Neighbors are connected to form a hull. The image also shows an isosurface extracted from the distance field to the hull.

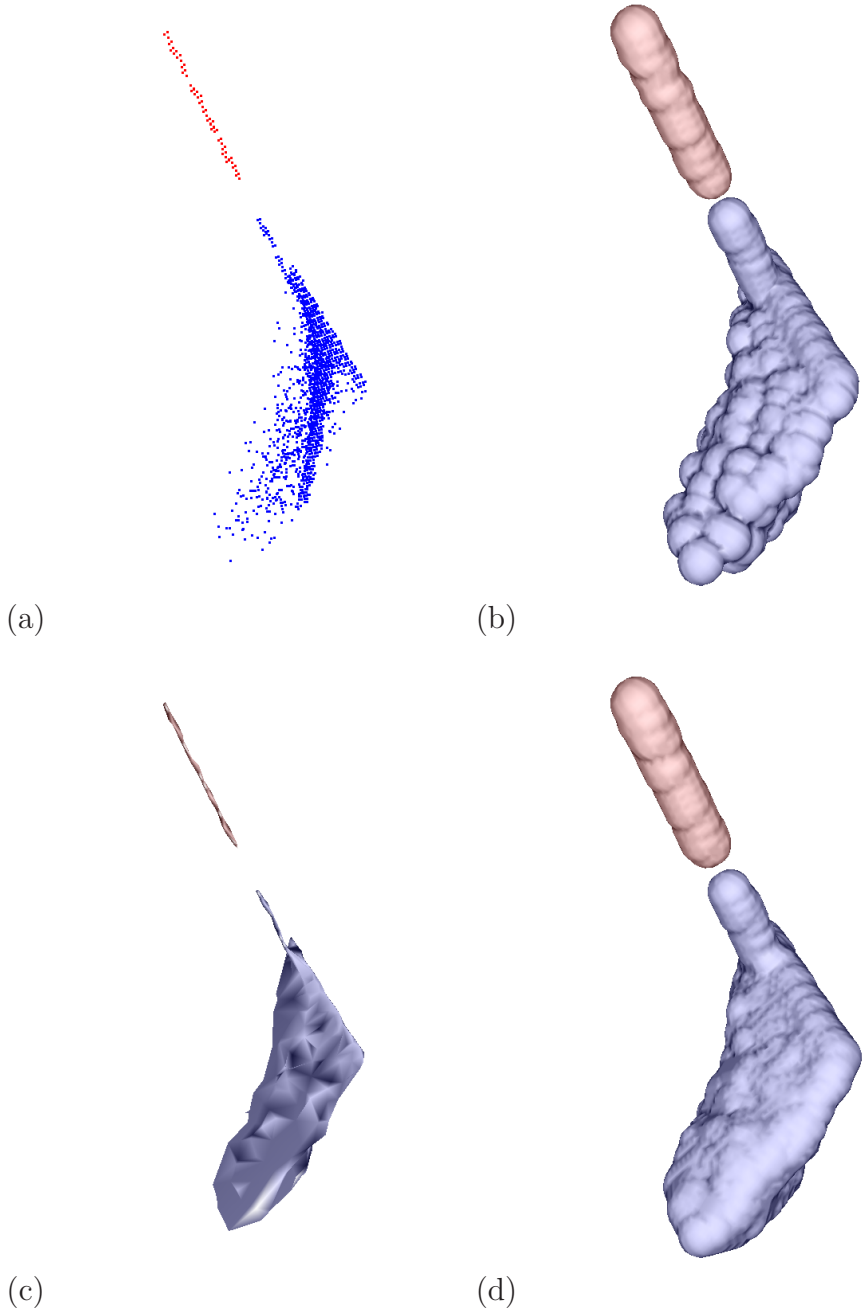
Additionally one can generate a distance field to the triangular mesh of the hull, again utilizing discrete Voronoi diagrams. By extracting an isosurface from this field, a third type of surface visualizing the shape of the cluster can be generated.

This approach leads to a smoother enclosing surface for the cluster points, which emphasizes the global shape of the cluster, as shown in Figure 2.23 (d).

This cluster visualization technique was applied to the points of the green cluster in Figure 2.21 (a). The distance fields were generated on a grid of size $64 \times 64 \times 37$. An isosurface, extracted with the standard marching-cubes algorithm [LC87] from the distance field to the hull of the cluster points, is shown in Figure 2.24.

In the second case of multi-variate non-spatial data the data points have no reference space, i. e. the data set represents no function but only a relation in $\mathbb{R}^m$. Nevertheless, a clustering of the data points can be computed in the same way as before. For the visualization of a cluster it is a priori not clear which space to use, since m might be much greater than 3. In such situations, the most common way is to project $\mathbb{R}^m$ to $\mathbb{R}^3$ and visualize the cluster in the projected space. For this purpose, we use a projection utilizing optimized star coordinates [LLRR08]. After projecting the whole data set to $\mathbb{R}^3$, we are again in the above situation and can use one of the proposed cluster visualization approaches.

We have shown, that the presented direct surface extraction techniques are also useful for the visualization of multi-dimensional and multi-variate data. After clustering the multi-variate data in feature space, the clusters can be directly visualized by extracting and displaying segmenting or bounding surfaces for volumetric data. In the case of multi-variate non-spatial data, the clusters are projected to $\mathbb{R}^3$ and visualized afterwards. The extracted surfaces give a good impression of the actual shape of the clusters and allow for a presentation in the context of the whole data set.



(a) (b)

(c) (d)

Figure 2.23: Different visualizations of two point clusters (colored red and blue) from the 2008 IEEE Visualization Design Contest data. The clusters were found using density-based clustering of the multidimensional feature space and were projected to a 3D visual space using a linear projection. Additionally to the cluster points (a), three types of enclosing surfaces are shown. (b) Isosurface extraction from distance field computed using a 3D discrete Voronoi diagram of resolution $256 \times 256 \times 256$. (c) Hull of the cluster computed from the isosurface of the distance field. (d) Isosurface extraction from the distance field to the hull in (c).

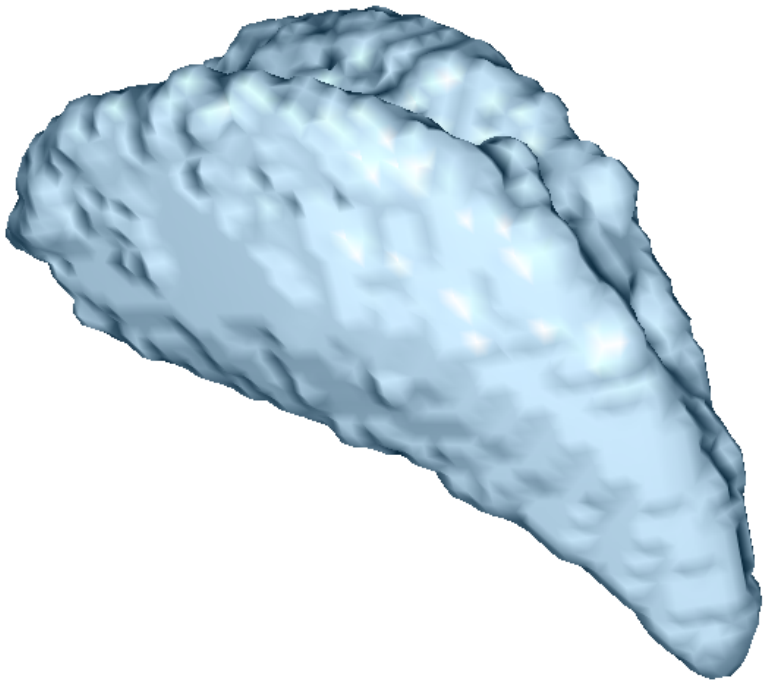


Figure 2.24: Rendering of the surface isodistant to the cluster hull of a smoothed particle hydrodynamics data set. The distance function was generated on a $64 \times 64 \times 37$ grid and the isosurface extracted using the marching-cubes algorithm.

Chapter 3

Level Sets

The direct isosurface extraction approach, described in Chapter 2, can result in poor results for noisy data sets or in sparsely sampled areas. Data sets typically exhibiting such characteristics are for example created by astrophysical smoothed particle hydrodynamics simulations. Astronomical objects like stars or nebulae are represented each by a set of particles holding several properties like temperature or density. The behavior of the particles is simulated over time while applying inner forces like pressure and gravity, but also outer forces like magnetic fields. The simulation is performed following a Lagrangian approach where also the particle positions in three-dimensional space change and each time step of the simulation is represented as an unstructured point-based volume data set. Examples of data sets with the described highly varying point density are given in Section 3.6.

The poor results of direct isosurface extraction in case of sparse sampling mainly result from the linear interpolation of isopoints between far apart sample points or in regions with fast changing function values. To avoid such negative effects one would rather want to extract isopoints from functions which are nearly linear and thus minimize errors in linear interpolation. The goal would be to find an auxiliary function that has nearly the same isosurface as the data set but is smooth and provides much better isopoints.

Since the direct calculation of a function fulfilling these conditions is practically unfeasible, the application of an approximation method is useful. Such an approximation method is the well-known and widely used level-set method introduced by Osher and Sethian [OS88].

The main idea of the classical approach is to represent surfaces in three-dimensional space as isosurfaces, the actual level sets, of an underlying scalar function, the static level-set function $\tilde{\varphi} : \mathbb{R}^3 \to \mathbb{R}$, and reshape these surfaces by deforming the function using partial differential equations (PDEs). Therefore, an auxiliary dimension is introduced, an artificial time. The dynamic level-set function

$$\varphi : \mathbb{R}^3 \times \mathbb{R} \to \mathbb{R}$$

is evolving over time with respect to the PDEs. This higher-dimensional perception

allows easy changes of the topology of the level sets and permits a huge variety of applications.

One standard task of the level-set method [OF03, Set99] is to deform the level sets with respect to an outer velocity vector field $\mathbf{V} : \mathbb{R}^3 \to \mathbb{R}^3$ representing external forces that affect the level sets. More precisely one has

$$\frac{\partial \mathbf{x}}{\partial t} = \mathbf{V} \tag{3.1}$$

for points $\mathbf{x} \in \mathbb{R}^3$ lying on a specified level set, i. e.

$$\varphi(\mathbf{x}, t) = \text{const} \ .$$

By differentiating this equation with respect to time t and using the chain rule, one gets

$$\frac{\partial \varphi}{\partial t} + \left\langle \frac{\partial \mathbf{x}}{\partial t}, \nabla \varphi \right\rangle = 0 \ .$$

This is the PDE modeling of the development of the level set. Together with Equation (3.1) one gets the final level-set equation

$$\frac{\partial \varphi}{\partial t} = - \left\langle \mathbf{V}, \nabla \varphi \right\rangle \ , \tag{3.2}$$

which describes the deformation of the level sets by the vector field $\mathbf{V}$.

This basic idea of level sets is commonly applied to gridded data [BWMZ05, MBZW02] and many variations and adaptions to different applications exist. However, none of the existing methods was able to directly operate on unstructured point-based data sets. We present a new approach, allowing for the direct application of the level-set method to unstructured point-based data sets without any grid calculation or reconstruction of the scalar field. The required function properties are directly computed from the data set and the level-set function is only processed at the positions of the data points.

We want to explicitly distinguish between our approach and particle level-set methods [ELF04, HK05], which have been presented in recent years. Actually these methods use free particles during the Lagrangian level-set computations. However, in contrast to our approach, particle level-set methods still require an underlying grid or mesh to compute the process of these particles. We present an approach that operates directly on unstructured point-based volume data.

The theoretical foundations of the described level-set-based approach are explained in detail in Section 3.1. As one can directly see from Equation (3.2) the calculation of derivatives of the level-set function is one key requirement of the method. Algorithms for approximating the gradient of the level-set function and mean curvature of the level sets have to be derived. These algorithms are presented in Sections 3.2 and 3.3.

To ensure a good behavior of the evolution process of the level-set function it has to be kept near a signed-distance function. All considerations concerning this problem

are presented in Section 3.4. A main aspect of the level-set approach is the evolution of the level-set function over time and its convergence. The time discretization and main aspects of convergence and stability of the whole process are discussed in Section 3.5. A comprehensive overview over the achieved results is given in Section 3.6, together with a discussion of performance and quality.

3.1 Theoretical Foundations

The main purpose of using level sets for isosurface extraction is to generate a surface that is smooth yet close to the real isosurface. To achieve this, an approach combining hyperbolic normal advection with mean curvature flow [GH86, Gra87, SHW05] is preferable.

Hyperbolic normal advection [OF03] models the attraction of the zero level set to the isosurface of the underlying scalar field in surface normal direction. More precisely the velocity vector field $\mathbf{V}$ in Equation (3.2) has the same direction as the surface normal, i. e.

$$\mathbf{V} = \lambda \cdot \mathbf{n} \,,$$

with a scalar normal velocity $\lambda : \mathbb{R}^3 \to \mathbb{R}$. Together with the fact that

$$\mathbf{n} = \frac{\nabla\varphi}{|\nabla\varphi|} \tag{3.3}$$

the level-set equation (3.2) becomes

$$\frac{\partial\varphi}{\partial t} = -\lambda\,|\nabla\varphi| \ .$$

At this point the actual data set f is needed again. The level-set function should be moved towards the scalar field $f - f_{\text{iso}}$, to achieve a coincidence of the zero level set with the isosurface to the isovalue f_{iso}. The normal velocity λ has to be chosen in a way that $|\varphi - (f - f_{\text{iso}})|$ is minimized, i. e. by choosing

$$\lambda = \varphi - (f - f_{\text{iso}}) \ .$$

This results in the level-set equation

$$\frac{\partial\varphi}{\partial t} = (f - f_{\text{iso}} - \varphi)\,|\nabla\varphi| \ . \tag{3.4}$$

A second property of the surface to achieve is smoothness, i. e. minimization of the surface area of the zero level set.

Let $\varphi : \mathbb{R}^3 \to \mathbb{R}$ be a differentiable function and Γ the zero level set of φ, i. e.

$$\Gamma := \left\{\mathbf{x} \in \mathbb{R}^3 \ : \ \varphi(\mathbf{x}) = 0\right\} \ .$$

The surface area of Γ can be obtained by

$$|\Gamma| := \int_{\mathbb{R}^3} \delta(\varphi)\,|\nabla\varphi|\,\mathrm{d}\mathbf{x}\;,$$

where δ denotes the one-dimensional Dirac δ-distribution [Bra00], defined by

$$\delta(x) := \lim_{h\to 0} \frac{1}{h\sqrt{\pi}} \mathrm{e}^{-\frac{x^2}{h^2}} \quad \text{for all} \quad x \in \mathbb{R}\;.$$

To successively minimize the surface area of Γ, the level-set function can be processed following the well-known model of mean curvature flow [ES91, ES92a, ES92b, ES95, ES98]. This leads to the level-set equation

$$\frac{\partial\varphi}{\partial t} = \kappa_\varphi\,|\nabla\varphi|\;, \tag{3.5}$$

where κ_φ denotes the mean curvature [MBW+05, Car76] of the level set, given by

$$\kappa_\varphi = \frac{1}{2}\left\langle \nabla, \frac{\nabla\varphi}{|\nabla\varphi|} \right\rangle\;. \tag{3.6}$$

Putting Equations (3.4) and (3.5) together, one gets a level-set model fulfilling both requirements for the smooth extraction of isosurfaces. This model can be represented by the equation

$$\frac{\partial\varphi}{\partial t} = \left(\mu\,(f - f_{\text{iso}} - \varphi) + (1-\mu)\kappa_\varphi\right)|\nabla\varphi|\;, \tag{3.7}$$

where the factor $\mu \in [0,1]$ controls the smoothness of the extracted surface.

3.2 Gradient Approximation

To process the level-set function φ following Equation (3.7), the gradient in each sample point has to be approximated. For the regular grid case, various finite differencing schemes [Bil04, Str89] exist which give very good results in short computing time. Unfortunately, these approaches cannot directly be applied to unstructured point-based volume data.

One possible approach to overcome this problem would be to locally interpolate the function values to a regular finite difference scheme. For this, one could use one of the well-known scattered data interpolation approaches and afterwards apply a finite difference scheme of choice. However, this would introduce resampling inaccuracies especially in regions with highly varying sample point density which are unwanted and can grow enormously.

Instead, it is favorable to generalize these finite difference schemes to directly operate on neighboring sample points. Such a generalization is derived in the following.

Definition 3.1 *Let $f : D \to \mathbb{R}$ be a scalar function on $D \subset \mathbb{R}^m$. The graph of f is the set* $\operatorname{graph}(f) \subset \mathbb{R}^{m+1}$ *defined as*

$$\operatorname{graph}(f) := \{(\mathbf{x}, f(\mathbf{x})) \ : \ \mathbf{x} \in D\} \ .$$

Definition 3.2 *Let $f : D \to \mathbb{R}$ be a differentiable scalar function on $D \subset \mathbb{R}^m$. The vector $\mathbf{n}_f(\mathbf{x}) \in \mathbb{R}^{m+1}$ is uniquely defined by its direction and length through the following two conditions.*

1. *$\mathbf{n}_f(\mathbf{x})$ is perpendicular to the tangent hyperplane $T_{(\mathbf{x},f(\mathbf{x}))}\operatorname{graph}(f)$ to $\operatorname{graph}(f)$ at the point $(\mathbf{x}, f(\mathbf{x}))$, i. e.*

$$\mathbf{n}_f(\mathbf{x}) \perp T_{(\mathbf{x},f(\mathbf{x}))}\operatorname{graph}(f) \ .$$

2. *The last component of $\mathbf{n}_f(\mathbf{x})$ equals -1, i. e.*

$$\left\langle \mathbf{n}_f(\mathbf{x}), \mathbf{e}^{m+1} \right\rangle = -1 \ ,$$

where $\mathbf{e}^i$, $i = 1, \ldots, m+1$ denotes the standard basis of $\mathbb{R}^{m+1}$.

With the help of the prior definitions it is now possible to formulate a proposition, which allows the inverse calculation of Equation (3.3). The gradient of a function can be found just by computing the surface normal to the graph of the function.

Proposition 3.3 *Let $f : D \to \mathbb{R}$ be a differentiable scalar function on $D \subset \mathbb{R}^m$. Then*

$$\nabla f(\mathbf{x}) = \operatorname{pr}_{\mathbb{R}^m} \left(\mathbf{n}_f(\mathbf{x}) \right) \ ,$$

where $\operatorname{pr}_{\mathbb{R}^m} : \mathbb{R}^{m+1} \to \mathbb{R}^m$ denotes the orthogonal projection to the first m coordinates and $\mathbf{n}_f(\mathbf{x})$ denotes the vector introduced in Definition 3.2.

Proof. Let $D \subset \mathbb{R}^m$, $f : D \to \mathbb{R}$ be a differentiable scalar function, and $(\mathbf{x}, f(\mathbf{x})) \in \operatorname{graph}(f)$. Furthermore let $\mathbf{e}^1, \ldots, \mathbf{e}^m$ be the standard basis of $\mathbb{R}^m$. The tangent hyperplane $T_{(\mathbf{x},f(\mathbf{x}))}\operatorname{graph}(f)$ to the submanifold $\operatorname{graph}(f)$ at the point $(\mathbf{x}, f(\mathbf{x}))$ is spanned by the vectors

$$\begin{aligned}
\mathbf{v}_i &= \frac{\mathrm{d}}{\mathrm{d}t} \left(\mathbf{x} + t\mathbf{e}^i, f\left(\mathbf{x} + t\mathbf{e}^i \right) \right) \big|_{t=0} \\
&= \left(\mathbf{e}^i, \frac{\mathrm{d}}{\mathrm{d}t} f\left(\mathbf{x} + t\mathbf{e}^i \right) \big|_{t=0} \right) \\
&= \left(\mathbf{e}^i, \left\langle \nabla f, \mathbf{e}^i \right\rangle \right) \\
&= \left(\mathbf{e}^i, \frac{\partial f}{\partial x_i} \right)
\end{aligned}$$

for $i = 1, \ldots, m$ [KN63, KN69]. Hence, a vector $\mathbf{v}$ is orthogonal to $T_{(\mathbf{x},f(\mathbf{x}))}\operatorname{graph}(f)$, iff

$$\mathbf{v} = c\left(\nabla f, -1 \right) \quad \text{for} \quad c \in \mathbb{R} \ .$$

Choosing $c = 1$ gives the assertion immediately. □

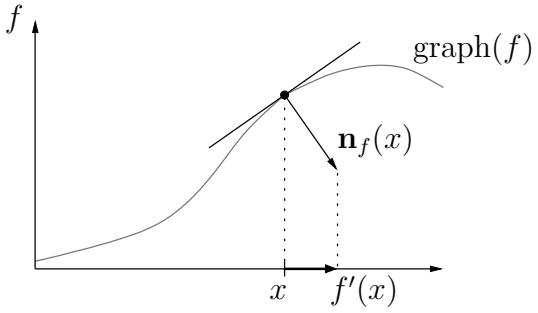


Figure 3.1: Relation between the rescaled normal to the graph and the derivation of a one-dimensional scalar function f.

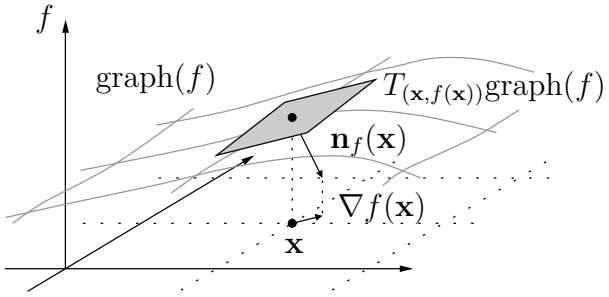


Figure 3.2: Graph of a two-dimensional function f with tangent plane $T_{(\mathbf{x},f(\mathbf{x}))}\text{graph}(f)$ and gradient $\nabla f(\mathbf{x})$ at a point $\mathbf{x}$.

One- and two-dimensional illustrations of these geometrical considerations are presented in Figures 3.1 and 3.2. From these pictures the idea of finding the gradient by calculating the normal vector to the graph of the function becomes very clear.

The proposition above gives an easy method for approximating the gradient of the level-set function φ by approximating the surface normal to the graph. The approximation of the surface normal is done using a standard least squares approach [LH95, MN03], fitting a linear function to the k nearest neighbors [Cle79, McN01] of each sample point.

To approximate the gradient of φ at the point $\mathbf{x}$ a three-dimensional linear function has to be fit to the k nearest neighbors of $\mathbf{x}$. The ansatz for $\mathbf{x}$ and each of the neighbors $\mathbf{y}^i$ is

$$a_1x_1 + a_2x_2 + a_3x_3 + a_4 = \varphi(\mathbf{x})$$

which leads to an overdetermined system of linear equations. The quadratic error of the system is minimized if

$$\mathbf{A}^T\mathbf{A}\mathbf{a} = \mathbf{A}^T\mathbf{b} \quad \text{with} \quad \mathbf{A} = \begin{pmatrix} \mathbf{x} & 1 \\ \mathbf{y}^1 & 1 \\ \vdots & \vdots \\ \mathbf{y}^k & 1 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} \varphi(\mathbf{x}) \\ \varphi(\mathbf{y}^1) \\ \vdots \\ \varphi(\mathbf{y}^k) \end{pmatrix}.$$

The matrix $\mathbf{A}^T\mathbf{A}$ is positive definite and symmetric, which allows us to use the

Cholesky decomposition to solve the system efficiently. The procedure finally can be subsumed by a closed formula for approximating the gradient.

Example 3.4 For a one-dimensional function $\varphi : \mathbb{R} \to \mathbb{R}$, represented through the points $(x_1, \varphi_1), \ldots, (x_k, \varphi_k)$, one gets

$$\frac{\mathrm{d}\varphi}{\mathrm{d}x} \approx \frac{k\sum_{i=1}^{k} x_i\varphi_i - \sum_{i=1}^{k} x_i \sum_{i=1}^{k} \varphi_i}{k\sum_{i=1}^{k} x_i^2 - \left(\sum_{i=1}^{k} x_i\right)^2} .$$

This formula is a generalization of several well-known differencing schemes. For example, using the points (x, φ_1) and $(x + h, \varphi_2)$ leads to the standard forward differencing scheme

$$\frac{\mathrm{d}\varphi}{\mathrm{d}x} \approx \frac{\varphi_2 - \varphi_1}{h} .$$

Choosing the points (x, φ_1), $(x+h, \varphi_2)$ and $(x-h, \varphi_0)$ leads to the central differencing scheme

$$\frac{\mathrm{d}\varphi}{\mathrm{d}x} \approx \frac{\varphi_2 - \varphi_0}{2h} .$$

□

The closed formula for approximating the gradient of a three-dimensional scalar function $\varphi : \mathbb{R}^3 \to \mathbb{R}$ can also be derived easily.

Example 3.5 Let the function φ be represented through the points $(x_i, y_i, z_i, \varphi_i)$ with $i = 1, \ldots, k$. Then the partial derivative in y-direction is approximated by

$$\frac{\partial\varphi}{\partial y} \approx \frac{X_1\sum_{i=1}^{k} x_i\varphi_i + X_2\sum_{i=1}^{k} y_i\varphi_i + X_3\sum_{i=1}^{k} z_i\varphi_i + X_4\sum_{i=1}^{k} \varphi_i}{Y} ,$$

where

$$\begin{aligned} X_1 = &\sum_{i=1}^{k} x_i y_i \left(k\sum_{i=1}^{k} z_i^2 - \left(\sum_{i=1}^{k} z_i\right)^2\right) \\ &+ \sum_{i=1}^{k} x_i z_i \left(\sum_{i=1}^{k} y_i \sum_{i=1}^{k} z_i - k\sum_{i=1}^{k} y_i z_i\right) \\ &+ \sum_{i=1}^{k} x_i \left(\sum_{i=1}^{k} y_i z_i \sum_{i=1}^{k} z_i - \sum_{i=1}^{k} y_i \sum_{i=1}^{k} z_i^2\right) , \end{aligned}$$

$$
\begin{aligned}
X_2 = &\sum_{i=1}^{k} x_i^2 \left(\left(\sum_{i=1}^{k} z_i \right)^2 - k \sum_{i=1}^{k} z_i^2 \right) \\
&+ \sum_{i=1}^{k} x_i z_i \left(k \sum_{i=1}^{k} x_i z_i - \sum_{i=1}^{k} x_i \sum_{i=1}^{k} z_i \right) \\
&+ \sum_{i=1}^{k} x_i \left(\sum_{i=1}^{k} x_i \sum_{i=1}^{k} z_i^2 - \sum_{i=1}^{k} x_i z_i \sum_{i=1}^{k} z_i \right) ,
\end{aligned}
$$

$$
\begin{aligned}
X_3 = &\sum_{i=1}^{k} x_i^2 \left(k \sum_{i=1}^{k} y_i z_i - \sum_{i=1}^{k} y_i \sum_{i=1}^{k} z_i \right) \\
&+ \sum_{i=1}^{k} x_i y_i \left(\sum_{i=1}^{k} x_i \sum_{i=1}^{k} z_i - k \sum_{i=1}^{k} x_i z_i \right) \\
&+ \sum_{i=1}^{k} x_i \left(\sum_{i=1}^{k} x_i z_i \sum_{i=1}^{k} y_i - \sum_{i=1}^{k} x_i \sum_{i=1}^{k} y_i z_i \right) ,
\end{aligned}
$$

$$
\begin{aligned}
X_4 = &\sum_{i=1}^{k} x_i^2 \left(\sum_{i=1}^{k} y_i \sum_{i=1}^{k} z_i^2 - \sum_{i=1}^{k} y_i z_i \sum_{i=1}^{k} z_i \right) \\
&+ \sum_{i=1}^{k} x_i y_i \left(\sum_{i=1}^{k} x_i z_i \sum_{i=1}^{k} z_i - \sum_{i=1}^{k} x_i \sum_{i=1}^{k} z_i^2 \right) \\
&+ \sum_{i=1}^{k} x_i z_i \left(\sum_{i=1}^{k} x_i \sum_{i=1}^{k} y_i z_i - \sum_{i=1}^{k} x_i z_i \sum_{i=1}^{k} y_i \right) ,
\end{aligned}
$$

and

$$
\begin{aligned}
Y = &\left(\sum_{i=1}^{k} y_i z_i\right)^2 \left(k\sum_{i=1}^{k} x_i^2 - \left(\sum_{i=1}^{k} x_i\right)^2\right) \\
&+ \left(\sum_{i=1}^{k} x_i z_i\right)^2 \left(k\sum_{i=1}^{k} y_i^2 - \left(\sum_{i=1}^{k} y_i\right)^2\right) \\
&+ \left(\left(\sum_{i=1}^{k} x_i y_i\right)^2 - \sum_{i=1}^{k} x_i^2 \sum_{i=1}^{k} y_i^2\right) \left(k\sum_{i=1}^{k} z_i^2 - \left(\sum_{i=1}^{k} z_i\right)^2\right) \\
&+ \sum_{i=1}^{k} y_i \sum_{i=1}^{k} z_i^2 \left(\sum_{i=1}^{k} x_i^2 \sum_{i=1}^{k} y_i - \sum_{i=1}^{k} x_i \sum_{i=1}^{k} x_i y_i\right) \\
&+ \sum_{i=1}^{k} x_i \sum_{i=1}^{k} z_i^2 \left(\sum_{i=1}^{k} x_i \sum_{i=1}^{k} y_i^2 - \sum_{i=1}^{k} x_i y_i \sum_{i=1}^{k} y_i\right) \\
&+ 2\sum_{i=1}^{k} x_i \sum_{i=1}^{k} x_i z_i \left(\sum_{i=1}^{k} y_i \sum_{i=1}^{k} y_i z_i - \sum_{i=1}^{k} y_i^2 \sum_{i=1}^{k} z_i\right) \\
&+ 2\sum_{i=1}^{k} x_i y_i \sum_{i=1}^{k} y_i z_i \left(\sum_{i=1}^{k} x_i \sum_{i=1}^{k} z_i - k\sum_{i=1}^{k} x_i z_i\right) \\
&+ 2\sum_{i=1}^{k} y_i \sum_{i=1}^{k} z_i \left(\sum_{i=1}^{k} x_i y_i \sum_{i=1}^{k} x_i z_i - \sum_{i=1}^{k} x_i^2 \sum_{i=1}^{k} y_i z_i\right) .
\end{aligned}
$$

Analogously, the partial derivatives in x- and z-direction can be obtained.

Just like in the one-dimensional case from Example 3.4, also the three-dimensional gradient approximation is a generalization of several well-known grid-based finite difference schemes. □

With the help of the considerations above, it is now possible to directly approximate the gradient of a scalar field given in form of an unstructured point-based data set and, hence, approximate the gradient terms of the level-set function in Equation (3.7).

Lemma 3.6 *The introduced least-squares gradient approach results in a consistent gradient approximation.*

Proof. For consistency, one has to verify that the approximated gradient in a point $\mathbf{x}$ converges towards the real gradient if the used points converge towards $\mathbf{x}$.

The proposed least-squares gradient approximation method uses the k nearest neighbors to approximate the gradient at a point $\mathbf{x}$. If the maximum distance of the used neighbors converges to 0, the computed hyperplane converges, by definition, towards the tangent hyperplane to the graph of the function. Together with Proposition 3.3, the approximated gradient converges towards the exact gradient at $\mathbf{x}$. □

The consistency is essential for the convergence of the whole level-set process and is obtained by using nearest neighbors for this approximation step instead of natural neighbors, cf. Chapter 2.

3.3 Mean Curvature Approximation

The processing of the level-set function following mean curvature flow, as modeled in Equation (3.7), requires the calculation of the mean curvature of isosurfaces at each sample point.

As already noted in Equation (3.6), the curvature κ_φ of the isosurface to a function φ can be expressed as

$$\begin{aligned}\kappa_\varphi :=& \frac{1}{2}\left\langle \nabla, \frac{\nabla\varphi}{|\nabla\varphi|}\right\rangle \\ =& \frac{1}{2}\left(\frac{\partial}{\partial x_1}\frac{(\nabla\varphi)_1}{|\nabla\varphi|} + \frac{\partial}{\partial x_2}\frac{(\nabla\varphi)_2}{|\nabla\varphi|} + \frac{\partial}{\partial x_3}\frac{(\nabla\varphi)_3}{|\nabla\varphi|}\right) .\end{aligned}$$

Here and in the following $(\nabla\varphi)_i$ denotes the ith component of the vector-valued function $\nabla\varphi$. This reduces the problem of calculating a second-order derivative to the calculation of four first-order derivatives, namely

$$\nabla\varphi\ , \quad \frac{\partial}{\partial x_1}\frac{(\nabla\varphi)_1}{|\nabla\varphi|}\ , \quad \frac{\partial}{\partial x_2}\frac{(\nabla\varphi)_2}{|\nabla\varphi|}\ , \quad \text{and} \quad \frac{\partial}{\partial x_3}\frac{(\nabla\varphi)_3}{|\nabla\varphi|}\ ,$$

at each sample point.

The process of approximating the mean curvature at each sample point is done in two steps. In a first step, the gradient of the level-set function φ, approximated using the algorithm from Section 3.2, is stored for each of the sample points.

Then the gradient to the three functions

$$\frac{(\nabla\varphi)_i}{|\nabla\varphi|} : \mathbb{R}^3 \to \mathbb{R}\ , \quad i = 1, 2, 3\ ,$$

is approximated using again the four-dimensional least-squares approach from above. The ith components of the gradients

$$\nabla\frac{(\nabla\varphi)_i}{|\nabla\varphi|} : \mathbb{R}^3 \to \mathbb{R}^3\ , \quad i = 1, 2, 3\ ,$$

are then taken and summed up to get an approximation for the mean curvature κ_φ. Altogether the approach needs three additional gradient calculations per sample point for the mean curvature approximation.

Example 3.7 As one can directly see from the considerations in Example 3.4, the method proposed here is again a generalization of several well known methods for grid-based approximations.

For a one-dimensional function $\varphi : \mathbb{R} \to \mathbb{R}$ represented by the three sample points $(x-h, \varphi_0)$, (x, φ_1) and $(x+h, \varphi_2)$, it leads to the standard central differencing

$$\varphi''(x) \approx \frac{\varphi_0 - 2\varphi_1 + \varphi_2}{h^2}$$

for the second order derivative. □

Because of using only multiple consistent gradient calculations, cf. Lemma 3.6, the consistency of the proposed method is obvious.

3.4 Reinitialization

The level-set method is in general only robust and efficient if several premises are fulfilled. One very important requirement is to keep the level-set function close to a signed-distance function to avoid overshooting effects [GF00, Kec98, PMO+99, SF99].

Definition 3.8 *A differentiable function $f : \mathbb{R}^m \to \mathbb{R}$ is called a signed-distance function to the $(m-1)$-dimensional submanifold $M \subset \mathbb{R}^m$, iff*

1. $f(\mathbf{x}) = 0 \quad \forall \mathbf{x} \in M$ *and*
2. $|\nabla f(\mathbf{x})| = 1 \quad \forall \mathbf{x} \in \mathbb{R}^m$.

The level-set function can be easily initialized as a signed-distance function. However, the level-set process following Equation (3.7) does not assure to maintain this property. In fact, the gradient of the level-set function is mostly diverging from a unit vector, resulting in a significant decrease of the quality in the level-set process.

Hence, it is necessary to remodel the level-set process to maintain the signed-distance property of the level-set function or reinitialize the level-set function to a signed-distance function if the length of the gradient exceeds a certain threshold.

The first strategy has come up in the last years [LXGF05] and differs significantly from the traditional approach. The introduction of an additional diffusion term, pulling the level-set function to a signed-distance function leads to an extra speed term in Equation (3.7), resulting in

$$\frac{\partial \varphi}{\partial t} = \lambda \operatorname{div}\left(\left(1 - \frac{1}{|\nabla\varphi|}\right)\nabla\varphi\right) + \left(\mu\left(f - f_{\text{iso}} - \varphi\right) + (1-\mu)\kappa_\varphi\right)|\nabla\varphi|$$

as level-set process equation. The parameter $\lambda \in \mathbb{R}^+$ weights the additional reinitialization term. To implement this method the calculation of an additional divergence

term would be necessary and with it the approximation of three additional gradients. The advantage of this approach is that no initial signed-distance function is needed. By choosing a sufficiently high λ each initial differentiable function is gradually converted to a signed-distance function.

For our implementation the traditional approach was chosen. The initial signed-distance function is processed following Equation (3.7), destroying the signed-distance property. After each level-set step, the norms of the level-set function gradients are calculated. If any of the norms exceeds a threshold, i. e. is no longer close to one, the function is reinitialized to a signed-distance function.

For the direct reinitialization of φ to a signed-distance function to the current zero level set, the closest distance of each sample point to the zero level set would have to be calculated [MBO94]. This is very time-consuming, especially for a high number of sample points and complex zero level set topologies.

Instead, the method of choice is the indirect level-set-based reinitialization, as proposed by Peng et al. [PMO+99]. The level-set function φ is processed following the special Eikonal equation

$$\frac{\partial \varphi}{\partial t} = \operatorname{sign}(\varphi) \left(\frac{1}{|\nabla \varphi|} - 1 \right) |\nabla \varphi|$$

until the process reaches steady state, i. e. until $|\nabla \varphi| \approx 1$.

One can directly see that this process maintains the zero level set and it converges very fast to a signed-distance function if the initial function was not far away. In fact, it is much faster than direct reinitialization.

3.5 Time Integration and Stability

In the previous considerations of level-set processing and reinitialization, several time evolution equations have been formulated. These are all of the general type

$$\frac{\partial \varphi}{\partial t} = F(\varphi) \, ,$$

where the operator F describes how the function $\varphi^t : \mathbb{R}^3 \to \mathbb{R}$ changes over time t. Since the function φ is not explicitely given, the change over time can only be calculated in discrete time steps. This raises the problem of choosing a time discretization for updating the function from one time step to the other and the related problem of convergence and stability of this process.

There exists a huge variety of time discretization schemes with different orders of accuracy, convergence speed, and prerequisites. Most commonly used are the Euler method [PTVF07], the TVD-Runge-Kutta approach [Shu88, GST01], and the WENO [JP00] and ENO schemes [OS91]. Each of these approaches can be easily applied to the methods described above, because they do not depend on the spatial

structure of the sample points. The time discretization chosen for the presented work is the explicit Euler method, i. e. the level-set function is processed following

$$\varphi^{t+\Delta t} = \varphi^t + \frac{\partial \varphi}{\partial t} \cdot \Delta t \; . \tag{3.8}$$

This choice of a first-order scheme is done for simplicity of the theoretical considerations concerning also the complex spatial derivative approximation and to allow an asynchronous update of the level-set function which is explained in Section 3.5.2.

3.5.1 Stability

One key requirement for evolution processes, as described in the previous sections, is convergence. If a solution to the given problem exists, the evolution process should generate a series of functions approximating the solution arbitrarily well. Without this property the application of such evolution processes is infeasible.

> The Lax-Richtmyer equivalence theorem [LR56, Str89] states that the convergence of a finite difference scheme is equivalent to consistency and stability. More precisely a discretized evolution process as in Equation (3.8) converges, iff the following two conditions hold:
>
> 1. The approximation of derivatives is consistent. More precisely, if the step size used for finite differences vanishes, the difference quotient converges against the derivative.
> 2. The discretized evolution process is stable, i. e. small errors in the function are not magnified during the process.

As explained in Sections 3.2 and 3.3, both introduced derivative approximation methods are consistent. The convergence of the whole evolution process is, hence, equivalent to its stability. Applying the finite difference schemes, derived in Sections 3.2 and 3.3, to Equation (3.8) leads to the general evolution equation

$$\varphi^{i+1}(\mathbf{x}) = (\mathbf{E}(\Delta t)\varphi^i)(\mathbf{x}) \tag{3.9}$$

for each sample $\mathbf{x} \in \mathbb{R}^3$. Here $\mathbf{E}$ is the discrete solution operator derived from the level-set equation and the derivative approximation schemes, which is dependent on the time step size Δt. This operator describes how the function values of time step $t + \Delta t$ are derived from the values of the function at time step t.

> In classical grid-based finite difference schemes the operator $\mathbf{E}$ is linear, i. e. a weakly filled matrix. In the observed elementary case of the proposed method it is very complex and highly non-linear as one can directly see from Example 3.5.

To study the stability of the time evolution of Equation (3.9), von Neumann stability analysis [LT05, Tho98] is used. This well-known analytical tool utilizes spatial

Fourier transforms [Bra00, DM72] to investigate if a finite difference scheme actually allows stability and in this case to find stable time step sizes Δt.

Von Neumann's theorem states that the evolution process (3.9) is stable with respect to the maximum norm, iff

$$\left|\tilde{\mathbf{E}}(\xi)\right| \leq 1 \,, \quad \text{for all} \quad \xi \in \mathbb{R} \,,$$

where $\tilde{\mathbf{E}}$ denotes the Fourier transform of $\mathbf{E}$.

The von Neumann stability analysis is performed on a two-dimensional example. This maintains simplicity but is complex enough to show the main problems and results.

Example 3.9 Let $a > 0$ be the normal speed of a moving interface in $\mathbb{R}^2$. The corresponding level-set model of hyperbolic normal advection is described by the level-set equation

$$\frac{\partial \varphi}{\partial t} = a|\nabla \varphi| \,.$$

Let further $(x, y) \in \mathbb{R}^2$ be a sample point in which the stability of the process should be investigated. The smallest reasonable number of neighbors used for gradient approximation is two.

Without loss of generality one can assume the coordinates (x, y, φ_1), $(x, y+1, \varphi_2)$, and $(x + \Delta x, y + \Delta y, \varphi_3)$ for the observed point and its two neighbors. In this case, a series of standard calculations [LT05, Tho98], applied to von Neumann's theorem leads to the following conditions on Δx, Δy, and Δt assuring stability

$$\lambda_y - \Delta y \frac{\lambda_x}{\Delta x} \leq \frac{1}{\Delta t} \tag{3.10}$$

$$\left((1 - \Delta y) \frac{\lambda_x}{\Delta x} + \lambda_y \right) \leq \frac{1}{\Delta t} \tag{3.11}$$

$$\frac{\lambda_x}{\Delta x} \geq 0 \tag{3.12}$$

$$\frac{\lambda_x}{\Delta x} \leq \frac{\lambda_y}{\Delta y} \,, \tag{3.13}$$

with

$$\lambda_x = \frac{a}{|\nabla \varphi|} \frac{\partial \varphi}{\partial x} \quad \text{and} \quad \lambda_y = \frac{a}{|\nabla \varphi|} \frac{\partial \varphi}{\partial y} \,.$$

It is obvious that Inequalities (3.10) and (3.11) can be maintained by choosing a small enough time step Δt, so they give only restrictions in the time dimension. Contrary to this, the last two conditions affect the relation between partial derivatives of φ and Δx, Δy.

Inequality (3.12) represents a concept well-known from classical level sets [OF03] called upwinding. It states that information should always propagate in the same

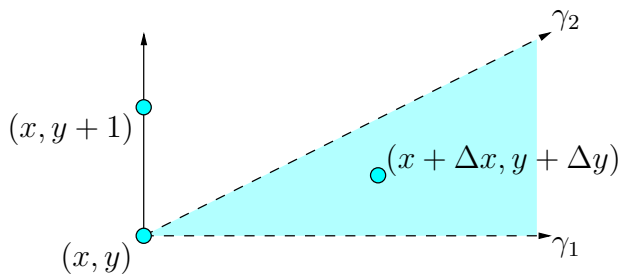


Figure 3.3: Illustration of the restriction to the position of the third sample point to allow stability of the level-set process. If the gradient in the point (x, y) is approximated with the help of the two points $(x, y+1)$ and $(x+\Delta x, y+\Delta y)$, the process is only stable if the point $(x+\Delta x, y+\Delta y)$ lies in the sector between the lines γ_1 and γ_2, where γ_1 is parallel to the x-axis and γ_2 has slope $\frac{\lambda_y}{\lambda_x}$.

direction as indicated by the waves of the level-set function. In our case this means that Δx and λ_x should always have the same sign.

Very similar but even more restrictive is Inequality (3.13). Together with Inequality (3.12) it ensures the upwinding in y-direction. Additionally it constitutes a relation between the ratio of both partial derivatives of φ and the ratio of Δx and Δy. More precisely it states that the point $(x + \Delta x, y + \Delta y)$ should always lie in the sector spanned by the line through (x, y) parallel to the x-axis and the line through (x, y) with slope $\frac{\lambda_y}{\lambda_x}$, as illustrated by Figure 3.3.

Altogether one can see that the proposed gradient approximation method allows stable time steps for the observed example. Therefore the sample points, used for the approximation of the gradient, have to have a specific upwind-like position. In this case, the biggest stable time step is explicitely given by Inequalities (3.10) and (3.11). □

As mentioned in Examples 3.4 and 3.5, the proposed gradient approximation method is a generalization of several well-known grid-based finite difference schemes. Hence, it is hardly surprising that the considerations in Example 3.9 lead to a generalization of the results of stability analysis for grid-based methods. Instead of allowing only axis aligned upwinding, it is possible to have the used sample points lying in a sector dependent on the partial derivatives of the level-set function.

All previous considerations can be easily extended to three dimensions. There it is possible to find similar regions of stability and in this case give restrictions to stable time steps.

Observing the previous considerations it is possible to ensure a stable process of the level-set function following normal advection by choosing appropriate neighbors and a small enough time step. However, this theoretical approach is not used in practice.

When practically solving the level-set equation for smooth isosurface extraction (3.7) it has turned out beneficial to use a much higher number of sample points for a very good approximation of spatial derivatives. The use of 26 nearest neighbors is also much more robust against the highly varying sample point density of some of the processed data sets. This procedure never ran into stability problems in practice when choosing small enough time steps Δt.

To derive an estimate for a time step Δt the considerations by Osher and Fedkiw [OF03] are adapted to the proposed method. The authors state a Courant-Friedrichs-Lewy-condition [CFL28, CFL67] necessary for the stability of the grid-based level-set method. The idea of the CFL-condition is to give a time step which allows stability by restricting the numerical domain of dependence of the finite difference scheme to the physical domain of dependence.

In fact, this condition is not sufficient for stability but is commonly used in practice. The adopted CFL-condition for Equation (3.7) becomes

$$\frac{\mu\,|f - f_{\text{iso}} - \varphi|}{d_{\min}|\nabla\varphi|}\left(\left|\frac{\partial\varphi}{\partial x_1}\right| + \left|\frac{\partial\varphi}{\partial x_2}\right| + \left|\frac{\partial\varphi}{\partial x_3}\right|\right) + \frac{6(1-\mu)}{d_{\min}^2} < \frac{1}{\Delta t}\,,$$

where $d_{\min}$ denotes the Euclidian distance to the nearest used neighbor, i. e. the radius of the minimal numerical domain of dependence.

Although there is no evidence that this CFL-condition ensures stability, the proposed level-set method can be carried out with time steps allowing stability in principle. This approach worked out very well for all practical cases considered. Indications for this and results from the procedure are shown in Section 3.6.

3.5.2 Asynchronous Time Integration

The considerations from Section 3.5.1 give a practicable time step to use, at least allowing stability for the process. However, all these observations are done in a fixed sample point at a certain point in time. To apply a global time step for all sample points, it is bounded by the most restrictive CFL-condition of all points of the data set. Thus, if the sample points have a highly varying distribution or if the underlying scalar field has big local variations, time steps for all points may be bounded by the rather restrictive stability condition of a few points. This behavior can dramatically slow down the whole level-set process.

To alleviate this potential drawback, one can use asynchronous time integration. Here, like in global time integration, all sample points start at the same point in time t_0. Then for each sample point one time step is computed, only bounded by the local stability condition. Most of the sample points are asynchronous afterwards, i. e. they have different time coordinates t.

To continue the level-set process for this asynchronous setting, a time line is introduced starting at t_0 and moving in positive time direction. Each sample point the time line reaches is again processed following the level-set equation with the local

bounded time step. To compute the next step for the actual sample point and approximate its derivatives, also function values of the neighboring points are needed. However, as the time line always proceeds in positive time direction the neighboring sample points are only equipped with their future function values.

To overcome this problem a local synchrony has to be generated. This can be achieved by using the applied Euler time integration. By storing for each sample point not only its current value and point in time but also the previous one, the function values to each point in time in between can be reconstructed by linear interpolation. This allows the approximation of gradients along the time line and with it the processing of the visited points.

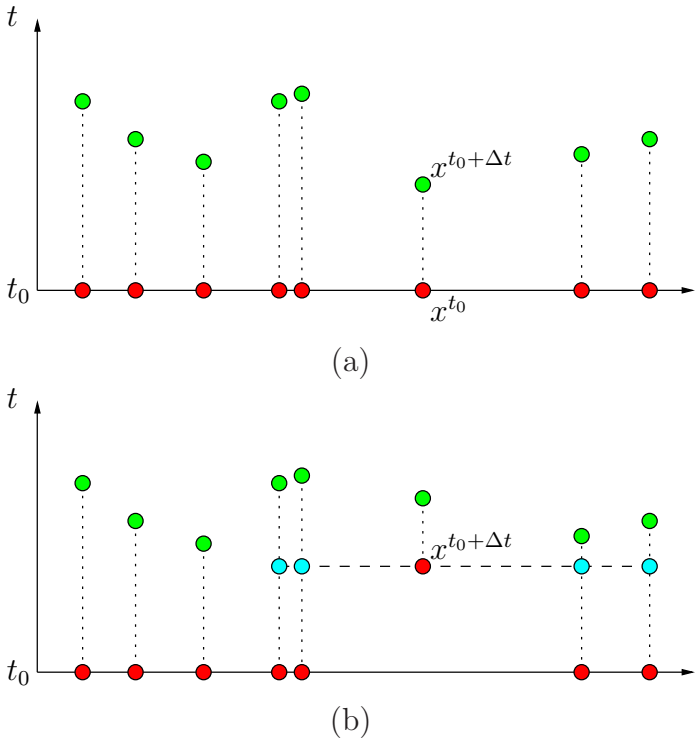


Figure 3.4: Asynchronous time integration of one-dimensional sample points. At the beginning of the process all sample points have the same evolution time t_0. After processing each point with its individual time step Δt, the sample points are asynchronous, as shown in (a). To process the asynchronous points, a time line is moving through time always proceeding with the point with the smallest time coordinate. The needed function values of neighboring points are obtained by linear interpolation with respect to the time, as illustrated in (b).

The asynchronous time integration process is illustrated in Figure 3.4. After having computed the first time step, the time line is set to the earliest subsequent point in time. The function values of the neighbors are linearly interpolated at this point in time. This interpolation is always possible, since always the earliest point in time is updated and all other points have values stored for a future point in time and a past point in time with respect to the current time line. With the interpolated

properties, the level-set function at the current sample point is integrated in time with the local time step restriction and the time line proceeds to the next point in time.

A detailed discussion of assets and drawbacks of the asynchronous time integration in comparison to the global time integration is given in the following section. There a detailed discussion of the usability with respect to actual data sets is given.

3.6 Results and Discussion

The presented level-set method was applied to a variety of different data sets. All experiments were performed on a single 2.66GHz Intel Xeon processor. To investigate the scalability of the approach it was first applied to a data set with different levels of detail. Therefore the regular Hydrogen data set was resampled at random sample positions and the function values were obtained by trilinear interpolation.

An illustration of the level-set process for the Hydrogen data set with four million sample points is shown in Figure 3.5. For each step of the level-set process, the zero level set was extracted. One can easily see how the topology of the zero level set changes and how smooth it converges against the desired isosurface.

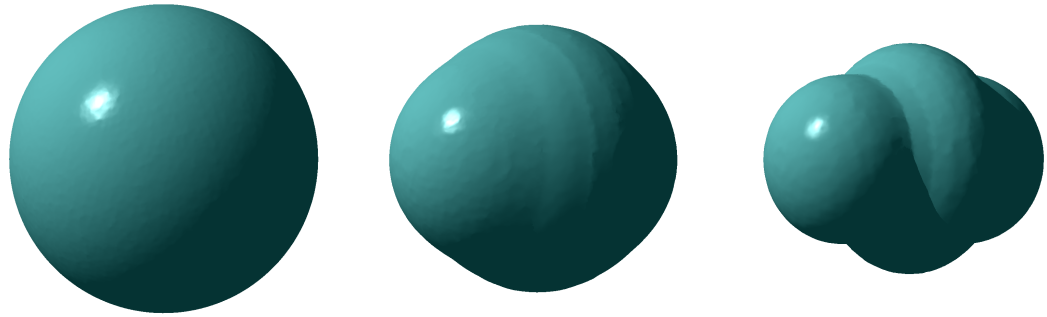
# samples	kd-tree generation	NN-calculation
500k	0.7 sec	23 sec
1,000k	1.9 sec	50 sec
2,000k	4.1 sec	111 sec
4,000k	9.0 sec	239 sec

Table 3.1: Computation times for the preprocessing of the Hydrogen data set with different sample quantities. The preprocessing includes the generation of the kd-tree and nearest neighbor (NN) calculation.

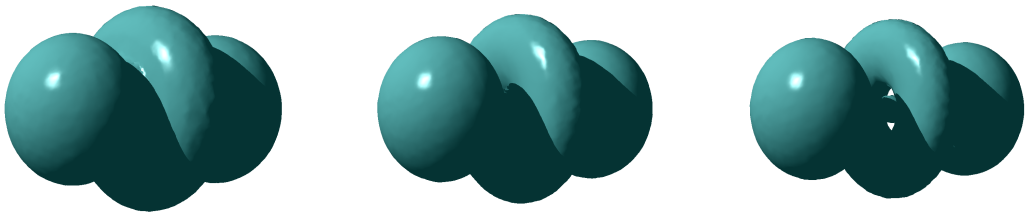
The performance of the preprocessing of the proposed level-set method is shown in Table 3.1. There the computation times for the kd-tree generation and for the computation of the 26 nearest neighbors are compared for different data set sizes. As expected the generation of the kd-tree as well as the nearest-neighbor calculation can be performed in $O(n \log n)$ time. As just a simple approach was implemented, the performance of the nearest-neighbor calculation could be surely improved by using more sophisticated algorithms.

As a second item of measurements, the computation times for the extraction of the zero level set were investigated. The level-set method was again applied to the Hydrogen data set, resampled with different sample quantities. After convergence of the level-set process, the zero level set was extracted using kd-tree-based direct surface extraction, as described in Chapter 2. The computation times are given in Table 3.2.

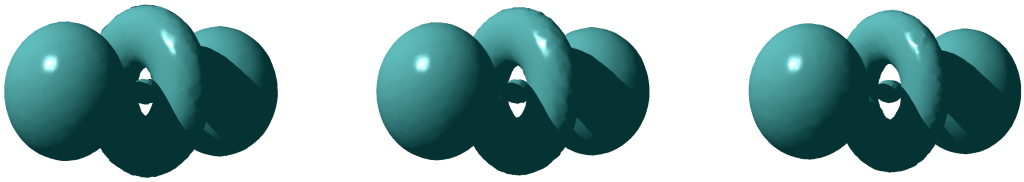
The neighborhood generation has a complexity of $O(n \log n)$, but is significantly faster than the calculation of the 26 nearest neighbors in Table 3.1. According to



Step 0 Step 1 Step 2



Step 3 Step 4 Step 5



Step 6 Step 7 Step 8

Figure 3.5: Splat-based ray tracing of the zero level sets generated during the level-set process. The level-set method was applied to the Hydrogen data set with four million randomly distributed sample points. The whole level-set process including preprocessing and extraction of the zero level set took 22 minutes. (Data set courtesy of Peter Fassbinder and Wolfgang Schweizer, SFB 382 University Tübingen.)

# samples	# isopoints	neighborhood	point extr.
500k	7k	0.8 sec	0.5 sec
1,000k	12k	1.7 sec	1.1 sec
2,000k	18k	4.1 sec	1.9 sec
4,000k	29k	10.1 sec	2.6 sec

Table 3.2: Times for the extraction of the zero level set for the hydrogen data set in different levels of detail. The number of extracted surface points and the computation times for the neighborhood computation as well as the isopoint extraction are shown. The zero level set was extracted after convergence of the level-set process.

this it is possible to follow the whole level-set process during runtime with just a little more computational effort.

	level-set process	reinitialization
synchr. int.	59k samples/sec	88k samples/sec
asynchr. int.	2k samples/sec	46k samples/sec

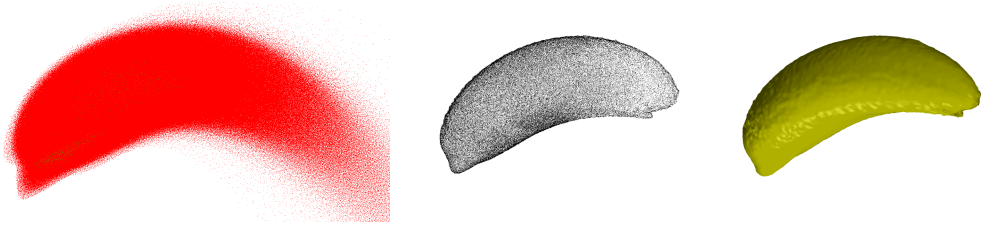
Table 3.3: Computation times comparison for the level-set as well as the reinitialization process with synchronous and asynchronous time integration. The times are specified in processed sample points per second.

To judge the computation times for the level-set process and the reinitialization is very difficult, since they heavily depend on the data set. For both synchronous and asynchronous time integration, it is possible to construct example data sets with arbitrarily long computation times due to sample points with nearly unstable behavior. That is why the computation times for this step are given in processed sample points per second, as presented in Table 3.3.

It is clearly visible that the asynchronous time integration heavily slows down the rate for the level-set process. This is mainly due to the fact that second derivatives have to be calculated. In the synchronous time integration this step uses the already calculated first derivatives, which are not available in the asynchronous approach. Hence, it can be very crucial for the overall computation time to choose the right approach for a data set. In terms of quality both time integration strategies are equal, as all performed experiments showed.

To show the practicability of the presented level-set approach, it was applied to a series of unstructured data sets coming from real applications. In this case, the data sets are time steps from astrophysical smoothed particle hydrodynamics simulations, provided by Stephan Rosswog from Jacobs University in Bremen. Each sample point holds a vector of different scalar values like density, temperature, and chemical mass fractions at this position.

First the smooth isosurface extraction pipeline is shown with the help of a SPH simulation data set with seven million unstructured sample points. In the simulations,



(a) (b) (c)

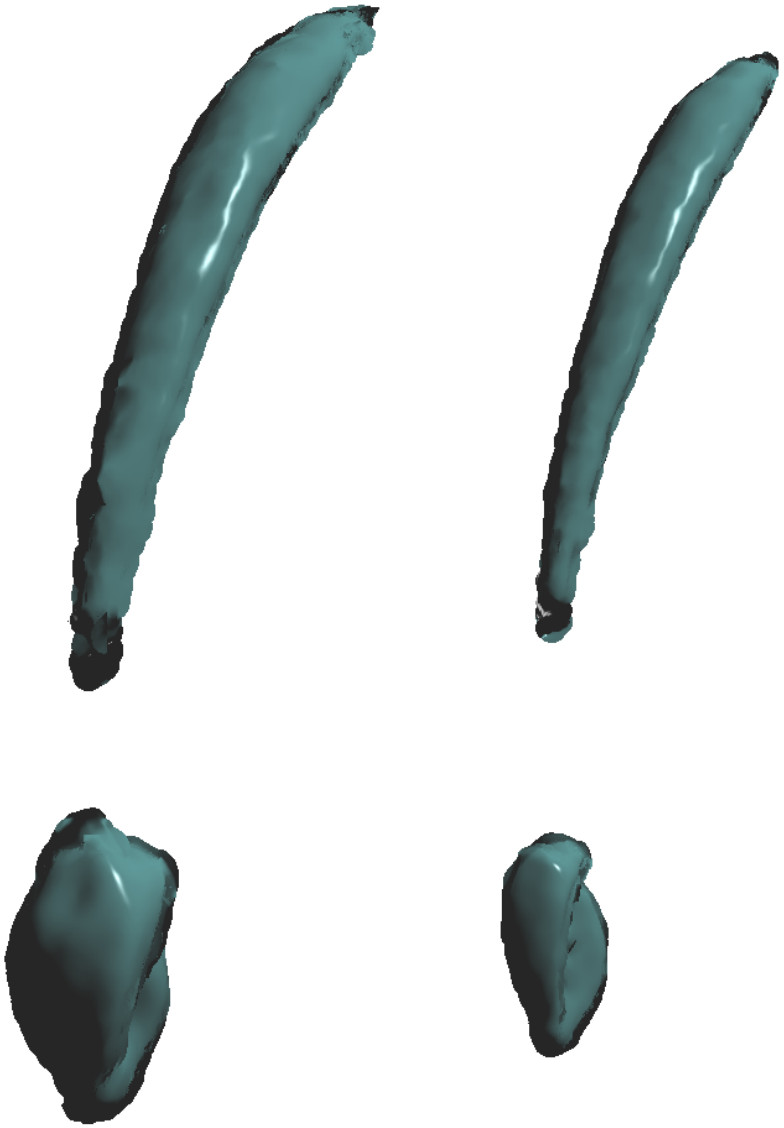
Figure 3.6: Smooth isosurface extraction pipeline in practice for an astrophysical SPH data set. The seven million unstructured sample points are rendered in (a) with colors related to their function values. After convergence of the level-set process the points on the zero level set are extracted, as shown in (b). The final rendering of the zero level set using splat-based ray tracing is visualized in (c).

the set of particles represents a white dwarf, i. e. a small and very dense star corpse. The white dwarf encounters a massive black hole and is torn apart. This results in drastically varying particle number densities, as shown in Figure 3.6 (a). In this image, some very far outlying particles in the very sparse lower right corner are even not drawn. The local point densities in these data sets can range over more than three orders of magnitude. After convergence of the level-set process, the extracted points on the zero level set are shown in Figure 3.6 (b). The final rendering of the zero level set can be seen in Figure 3.6 (c).

Two different isosurfaces have been extracted with respect to the density scalar field from another white dwarf data set with 500,000 sample points. The smooth isosurfaces obtained by using the asynchronous level-set approach are shown in Figure 3.7. The whole process for generating both isosurfaces lasted 68 minutes. Using the synchronous approach would have been not feasible, since the very small time steps in this data set are mainly bound by at most 20 sample points, leading to several hours of computation time.

On the same data set the quality of the level-set approach compared to direct isosurface extraction was investigated. A side-by-side comparison for the same isovalue of the density field is shown in Figure 3.8. The isopoints are rendered as small discs, which directly reveal the much higher quality and accuracy of the isosurface extracted by the level-set approach. A close-up view shows some far outliers for the direct isosurface extraction, which result from the highly varying point density of the data set.

Finally, we tested our presented derivative approximation methods in terms of accuracy. We applied our method to synthetic unstructured point-based data sets representing several three-dimensional test functions, including polynomials and rational functions up to order five and trigonometric functions. The data sets consist of one million sample points randomly distributed within $[0, 100] \times [0, 100] \times [0, 100]$. The approximated derivatives, as well as derivatives obtained by central differencing, were compared to the real analytic derivatives. The approximated gradients



$\rho = 40\text{g/cm}^3$ $\rho = 100\text{g/cm}^3$

Figure 3.7: Splat-based rendering of two different isosurfaces of the 500k white dwarf simulation data set. The underlying scalar field represents the density in space. Smooth isosurfaces were extracted for two different densities using asynchronous time integration.

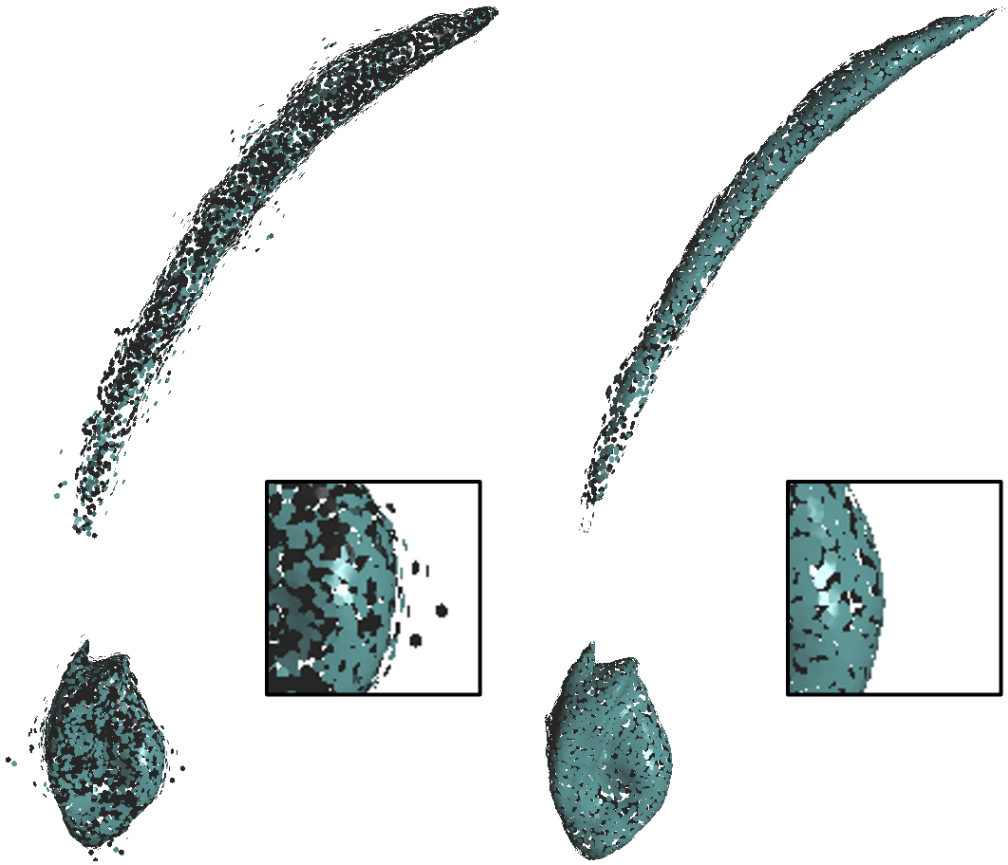


Figure 3.8: Comparison between direct isosurface extraction on the left side and smooth isosurface extraction using level sets on the right side for the 500k white dwarf simulation data set. To illustrate the significant advantage of the level-set approach over direct isosurface extraction, a close-up view on the surface and a rendering of the surface points with very small splats is shown.

were computed with the 26 nearest neighbors of each sample point. For central differencing the needed sample points were computed from the given sample points by stepping in the direction of the axes with step size 1. The function values at these points were taken from the analytical function, not by using scattered data interpolation. For each data point, the relative error δ between exact gradient $\nabla\varphi$ and approximated gradient $\tilde{\nabla}\varphi$ was computed by

$$\delta = \frac{\left|\nabla\varphi - \tilde{\nabla}\varphi\right|}{|\nabla\varphi|} .$$

The average relative error was calculated by computing the arithmetic average of the relative errors at the data points.

When using polynomial test functions, the average relative errors of central differencing and our approximation method are equal. Depending on the order of the polynomial the average relative error is between $2 \cdot 10^{-7}$ and $9 \cdot 10^{-7}$. When dealing with rational and trigonometric functions, the errors may differ. The average relative error for central differencing lies between $1 \cdot 10^{-5}$ and $4 \cdot 10^{-5}$, while the average relative error for our approximation method ranges from $2 \cdot 10^{-5}$ to $13 \cdot 10^{-5}$. Hence, the proposed gradient approximation technique is close to standard techniques in terms of accuracy and well suited for derivative approximations from unstructured point-based volume data.

We have presented a level-set approach, which is the first one directly able to be applied to unstructured point-based volume data. Since no competitive techniques exist, we consider approaches with similar intention. Museth et al. [MBZW02] propose a method for applying level sets to multiple non-uniform data sets, which consist of unions of rectangular grids. Our results are comparable in terms of quality with the ones presented there. Additionally it would be easy to process the non-uniform data used there with our approach. Unfortunately, the authors state no computation times for their examples.

State-of-the-art level-set approaches for regular grids are able to process up to one million grid points per second [MBNM07], which would be one order of magnitude faster than our approach for unstructured point-based data. However, the interpolation of the data to a hexahedral grid is no option for highly varying data sets, due to the introduced interpolation errors.

Chapter 4

Point-cloud Rendering

The second step of a visualization pipeline after preprocessing and transforming the data is mostly the rendering of generated virtual models. This is a very crucial part, since the preprocessed data has to be visualized on the screen or any other output device in a way that users can easily see important properties and features of the data.

The results of the first part of the presented visualization pipeline for unstructured point-based volume data are surfaces in point-cloud representation. These surfaces represent isosurfaces with specific properties of the underlying scalar field. Since no connectivity information is available for the surface points and it would be unfeasible to obtain it during isopoint extraction, standard rendering techniques using meshes are not directly applicable.

Nowadays, different point-based rendering approaches exist [AGP+04, LP01, PZvBG00, RL00, SPL04]. The most prominent ones are based on the local approximation of the surface by fitting planes or polynomial surfaces to a subset of neighboring points [ABCO+01]. Technically, these are still (local) surface reconstructions. However, the steps for extracting the surface points are mainly point-based without reconstruction of fields or structures. So it is also preferable to not reconstruct the surface from the surface points but use a point-cloud rendering.

Whatever method for rendering is used, the knowledge of surface normals at least at the surface points is essential to facilitate shading. From the surface extraction techniques, described in the previous chapters, only an orientation vector is given for each surface point. A surface normal at each point is approximated in a preprocessing step using an approach based on principal component analysis. A detailed description of this step is given in Section 4.1.

For the actual rendering step, two choices are presented. If a photorealistic rendering with global illumination and effects like shadows, reflections, or transparency is desired a splat-based ray tracing approach is used [LMR07]. Since many results are rendered using this method, it is explained briefly in Section 4.2.

For the direct rendering of point clouds without any precomputations an image-space point-cloud rendering method is proposed [RL08a]. The approach, explained in de-

tail in Section 4.3, uses image-space operations and filters to compute high-quality renderings. By using today's graphics card capabilities this approach achieves interactive frame rates. A number of experiments and examples and a detailed discussion of both approaches are given in Section 4.4.

4.1 Surface Normal Approximation

In the previous chapters, a series of different surface extraction techniques for unstructured point-based volume data is proposed. The results of all approaches are surfaces in point-cloud representation. More precisely the surfaces are represented by a set of volumetric points lying on the surface without any connectivity or neighborhood information. The rendering of such point clouds requires at least additional knowledge about surface normals at the surface points to allow shading. However, the presented approaches are not able to provide these. The only additional information besides the position of the surface point in three-dimensional space is a vector giving an orientation of the surface.

To obtain the desired surface normals a standard approximation approach [PKG03] using principal component analysis [FP02] is chosen. Although this strategy was proven to be not the best in terms of accuracy by Alexa et al. [AA04], it is very fast and gives good results.

For each surface point $\mathbf{x} \in \mathbb{R}^3$, a number of surrounding surface points have to be found. Typically, n nearest neighbors $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^3$ are chosen. In the presented approach, a number of $n = 30$ nearest neighbors produced always good results and was fast to compute with the help of a three-dimensional kd-tree. Afterwards the centroid $\bar{\mathbf{x}}$ has to be calculated from the set $\{\mathbf{x}, \mathbf{x}_1, \ldots, \mathbf{x}_n\}$, i. e.

$$\bar{\mathbf{x}} := \frac{1}{n+1} \left(\mathbf{x} + \sum_{i=1}^{n} \mathbf{x}_i \right) .$$

A principal component analysis of this set with respect to the centroid is done calculating first the matrix of difference vectors

$$\mathbf{A} := \begin{pmatrix} \mathbf{x} - \bar{\mathbf{x}} \\ \mathbf{x}_1 - \bar{\mathbf{x}} \\ \vdots \\ \mathbf{x}_k - \bar{\mathbf{x}} \end{pmatrix} .$$

The respective covariance matrix $\mathbf{C} := \mathbf{A}^T\mathbf{A}$ is positive semidefinite and symmetric. Thus, all eigenvalues, which are used for principal component analysis, are real and greater or equal than zero. The eigenvalues λ_i are calculated using the standard approach, solving

$$\det(\mathbf{C} - \lambda\mathbf{I}) = 0 ,$$

where $\mathbf{I}$ denotes the identity matrix. This leads to an equation with a polynomial of degree three. The roots λ_i of this polynomial can be explicitely calculated by Cardano's formula [Dun91].

The eigenvectors of $\mathbf{C}$ are orthogonal to each other. The eigenvector to the largest eigenvalue gives the direction with the biggest extension of the point set. The eigenvector to the second largest eigenvalue gives the orthogonal direction with the second biggest extension of the point set and so on. The eigenvector to the smallest eigenvalue gives, consequently, the direction perpendicular to the largest extensions of the point set.

Since all points lie on one surface the eigenvector to the smallest eigenvalue is a good approximation for the normal vector to the surface. If $\tilde{\lambda}$ is the smallest eigenvalue of $\mathbf{C}$, an associated eigenvector $\mathbf{v} \in \mathbb{R}^3$ can be calculated by finding a nontrivial solution of

$$\left(\mathbf{C} - \tilde{\lambda}\mathbf{I}\right)\mathbf{v} = \mathbf{0}\ .$$

This eigenvector is afterwards normalized and oriented with respect to the given surface orientation to obtain the approximated surface normal.

The used approximation method gives very good results in short computation times, especially for big sets of surface points. A comprehensive analysis of the computation times and results are presented in Section 4.4. Even though the chosen approach for approximating the surface normals can give results of lower quality in some special cases, it has worked out very well for the purposes pursued here.

From now on we can assume that the surface Γ to be rendered is given in a point-cloud representation $\tilde{\Gamma}$ with associated normals, i. e.

$$\tilde{\Gamma} := \left\{(\mathbf{x}_i, \mathbf{n}_i) \in \Gamma \times T_{\mathbf{x}_i}\Gamma^{\perp} \ :\ \|\mathbf{n}_i\| = 1\right\}\ ,$$

where $T_{\mathbf{x}_i}\Gamma^{\perp}$ denotes the orthogonal complement to the tangent plane $T_{\mathbf{x}_i}\Gamma$ to the surface Γ at point $\mathbf{x}_i$.

4.2 Splat-based Ray Tracing

Ray tracing is a well-known and widely used rendering technique in computer graphics. It allows for precise shadow computations and modeling of light reflection and refraction. The original idea [App68, Whi80] of ray tracing of triangular meshes has a long tradition in photorealistic rendering and is described in any computer graphics textbook, e. g. [Wat00]. For each pixel of the screen a ray is shot from the view point through the center of the pixel and traced through the scene. The color of the pixel is calculated with respect to the properties of possibly hit surfaces and light sources.

With the upcoming of precise high-resolution 3D laser scanning techniques, point clouds have gained major interest in the computer graphics society. Several well-known rendering techniques have been remodeled to be applicable to surfaces in point-cloud representation. This is also the case for ray tracing. Schaufler and Jensen [SJ00] were the first to propose a ray-tracing technique for point clouds

based on the idea of sending out rays with a certain width which can geometrically be described as cylinders. This approach does not handle varying point density within the point cloud. Moreover, the surface generation is view-dependent, which may lead to artifacts during animations. Wand and Straßer [WS03] introduced a similar concept by replacing the cylinders with cones.

Adamson and Alexa [AA03] proposed a method for ray tracing point set surfaces. For the intersection of the rays with the locally reconstructed surfaces, points on the ray are iteratively projected onto the surface until the procedure converges, which is computationally very intense. An interactive ray tracing algorithm of point-based models was introduced by Wald and Seidel [WS05]. They propose a splat-based hybrid approach which uses ray tracing for shadow computation. The actual shading is performed using local shading models. Thus, transparency and mirroring reflections are not modeled.

The approach presented in the following, allows for the generation of such ray-tracing-specific properties. It is divided into two steps. The first step is based on the ideas of surface splatting [ZPvBG01] and described in Section 4.2.1. For each surface point with surface normal a radial expansion tangential to the surface is computed. These generated discs are called splats. They are supposed to overlap in order to cover the entire surface. Furthermore a local normal field has to be computed per splat depending on the local curvature to achieve a smooth looking surface.

The actual ray-tracing step is executed by sending out rays that intersect the splats, potentially being reflected or refracted. Surface normals are interpolated from the normal fields and between splats, where they overlap. An octree is used to improve the computation times and minimize the ray-splat intersection calculations. A more detailed description of this step is given in Section 4.2.2. Since the splat generation is view-independent, it can be carried out as a preprocessing step. For animations only the ray-tracing step has to be done per animation frame.

4.2.1 Splat Generation

Let P be a point cloud consisting of n points with normals $(\mathbf{p}_1, \mathbf{n}_1), \ldots, (\mathbf{p}_n, \mathbf{n}_n) \in \mathbb{R}^3 \times \mathbb{R}^3$. A set of m splats $S_1, \ldots, S_m$ has to be generated that covers the entire surface represented by the point cloud P. For each of these splats its radius $r_i \in \mathbb{R}$, $i = 1, \ldots, m$, and a normal field $\mathbf{n}_i(u, v)$, $i = 1, \ldots, m$ has to be calculated, where $(u, v) \in [-1, 1] \times [-1, 1]$ with $u^2 + v^2 \leq 1$ describes a local parameterization of the splat.

The radii of the splats should vary with respect to the curvature of the surface covered by the splat. In regions of high curvature, a piece-wise constant surface representation via splats requires us to use many splats with small radii to stay within a predefined error bound. The error for a surface point is measured as the distance perpendicular to the splat's plane.

The algorithm for generating the splats with respective normal fields goes sequentially through the list of surface points. Through each non-visited surface point $\mathbf{p}$

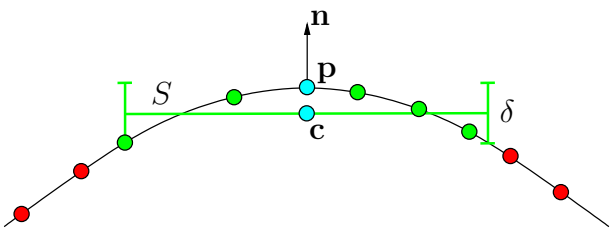


Figure 4.1: Generation of the splat S for the point $\mathbf{p}$ in side view. The splat is grown in a plane perpendicular to the normal $\mathbf{n}$ as long as the distance to the covered points, colored green, stays in a given threshold δ. Afterwards the center point $\mathbf{c}$ of the splat is set in a way that the maximum distance of the covered points to the splat is minimized.

a plane is fit perpendicular to the surface normal. The neighboring surface points are sorted with respect to their Euclidean distances to $\mathbf{p}$. Next, a splat S lying in the plane and centered at $\mathbf{p}$ is grown iteratively as long as it only covers neighbors within the given error bound δ. At each iteration step, the radius of the splat is increased, such that it covers one additional neighbor of $\mathbf{p}$. The normal of the splat remains unchanged, but the center $\mathbf{c}$ is moved along the surface normal $\mathbf{n}$ such that the splat position minimizes its maximal distance to all so far covered points. After the iteration stops, the maximum radius is saved for the splat and its midpoint $\mathbf{c}$ is set to minimize the maximum distance of the splat to all covered surface points. The process of generating the splat for one surface point $\mathbf{p}$ with normal $\mathbf{n}$ is illustrated in Figure 4.1.

After each generation of a splat for a surface point $\mathbf{p}$ a number of other surface points is covered by the splat of $\mathbf{p}$. From these points not all have to be still considered for splat generation. The amount of splats needed to cover the surface represented by the point cloud P depends on the chosen error bound. Which splats to generate and how many is a non-trivial task [WK04]. Here a simple heuristic based on the relative distances to the splats' centers is used.

If a splat with radius r is generated starting from the point $\mathbf{p}$, then no splats need to be generated starting from neighboring points within the projected distance $\lambda \cdot r$ from the splat's center $\mathbf{c}$. Here $\lambda \in [0, 1]$ is the factor that defines the percentage of the splat's radius used for the criterion. It is defined globally for P, which is possible as it is multiplied with the locally varying radius r. A good choice for λ is a value such that the generated splats cover the entire surface and have low overlap, which is very tough to guarantee. The parameter choice for a data set depends on the variance of the local curvature in combination with the variance of the local point density. However, in most of the experiments and all shown pictures in Section 4.4, a choice of $\lambda = 0.2$ gave very good results.

In order to generate a smooth-looking visualization of a piecewise linear surface, the normals have to be smoothly interpolated over the surface before applying the light and shading model. This is done by generating a linearly changing normal field

within each splat, cf. [BSK04].

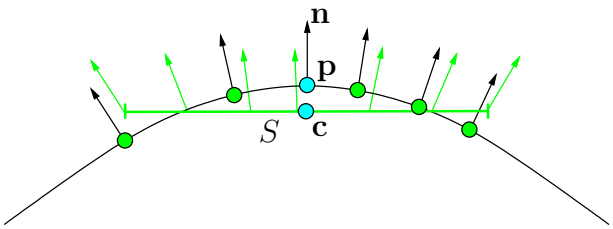


Figure 4.2: Generation of linear normal field (green) over the splat S from the normals of the points covered by the splat. The normal field is generated using local parameters $(u, v) \in [-1, 1]^2$ over the splat's surface. The normal at $(0, 0)$ may differ from $\mathbf{n}$.

Let $S = (\mathbf{c}, \mathbf{n}, r)$ be a splat generated as described above. For the definition of the normal field over S, the splat is parameterized locally using two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^3$ with $\mathbf{u} \perp \mathbf{n}$ and $\mathbf{v} = \mathbf{n} \times \mathbf{u}$. Moreover, let $\|\mathbf{u}\| = \|\mathbf{v}\| = r$, i. e. the local parameterization of the splat is given by

$$(u, v) \mapsto \mathbf{c} + u \cdot \mathbf{u} + v \cdot \mathbf{v}$$

with $(u, v) \in \mathbb{R}^2$ and $u^2 + v^2 \leq 1$. Using this local parameterization, a linearly changing normal field $\mathbf{n}(u, v)$ can be defined by

$$\mathbf{n}(u, v) = \mathbf{n} + u \cdot \lambda_u \cdot \mathbf{u} + v \cdot \lambda_v \cdot \mathbf{v} \,, \tag{4.1}$$

with parameters $\lambda_u, \lambda_v \in \mathbb{R}$. These parameters have to be chosen to fit the normals of the covered surface points as close as possible. An illustration of the idea is given in Figure 4.2.

The factors λ_u and λ_v are approximated by a least-squares fitting approach. The normal and position of each covered surface point is projected onto the splat. With the help of its local coordinates Equation (4.1) is formed leading to a system of linear equations with unknown variables λ_u and λ_v for all covered points. This system is overdetermined and a solution can be approximated in the least-squares sense. To allow more flexibility also the normal $\mathbf{n}$ in $\mathbf{c}$ is set to be unknown.

Splat and normal field generation are done in a preprocessing step. For each splat only the center point $\mathbf{c}$, the center normal $\mathbf{n}$, the local parameterization vectors $\mathbf{u}$ and $\mathbf{v}$, and the normal field parameters λ_u and λ_v have to be stored. The actual point cloud is not needed any further.

4.2.2 Ray Tracing

The input of the ray-tracing procedure are the splats $S_1, \ldots, S_m$, each given by its center $\mathbf{c}_i$, its radius r_i, and its normal field parameters. The ray-tracing method sends out primary rays from the camera position through the center of each pixel of the resulting image onto the scene. The intersections of the primary rays with the surfaces

are computed using ray-splat intersections. From these intersection secondary rays, i. e. shadow rays, reflection rays, or refraction rays, are sent out depending on the surface properties. In the latter two cases, the rays are treated equally to primary rays and traced recursively until the ray-trace depth is reached. The results of the ray computations are combined using the Blinn-Phong lighting model [Bli77].

In order to process computations of ray-splat intersections efficiently, an octree is used for storing the splats. The generation of this octree and the insertion of the splats is done in two steps.

The first step is the dynamic phase, where the actual octree is generated using a standard approach [Sam06]. Starting with an empty octree that describes the bounding box of the entire scene, each splat's center point is inserted iteratively in the respective leaf of the octree. As soon as a leaf of the octree contains more points than a given threshold, it gets split into eight equally sized subcells. The points are inserted into the new leaves with respect to their positions in space. The iteration stops once all splat center points have been inserted.

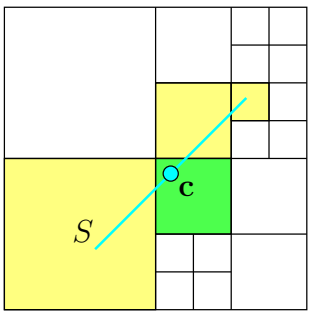


Figure 4.3: Insertion of splats into the octree. In a first step, the octree is built by just inserting the midpoint $\mathbf{c}$ of each splat. In a second step, the structure of the octree remains unaffected. Each splat S is, additionally to the cell of its midpoint (green cell), inserted into all cells it intersects (yellow cells).

In the second step, splats are inserted into further cells of the octree but its shape remains static, i. e. no further cell subdivisions are executed. These additional splat insertions are necessary, as splats have an expansion and may stretch over various cells. Thus, splats have to be additionally inserted into all leaf cells they intersect, as illustrated by Figure 4.3.

Since an exact cell-splat intersection is computationally rather expensive, a nested test for potentially intersecting cells is done for each splat S. The first test checks which cells intersect the bounding box with length $2 \cdot r$ centered at the center point $\mathbf{c}$ of the splat. For all leaf cells, for which the first test was positive, a second test based on the local parameterization of the splat is done. The cells are checked for intersections with the bounding square of the splat, spanned by the local parameterization vectors $\mathbf{u}$ and $\mathbf{v}$. If a cell passes both tests, the splat is inserted into it. This nested test is very fast and each splat is only stored slightly more often than

necessary. After the whole process, each splat is recorded in all leaves of the octree which cells it intersects.

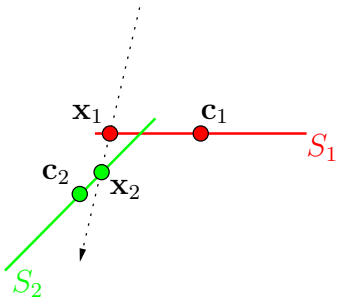


Figure 4.4: Intersection of rays with splats. The ray intersects both splats S_1 and S_2 in the points $\mathbf{x}_1$ and $\mathbf{x}_2$ respectively. $\mathbf{x}_2$ is chosen as intersection point since the relative distance to its splat's center $\mathbf{c}_2$ is the smallest one.

The actual intersection of rays with splats is computed using the octree partitioning of the three-dimensional scene. The primary rays are traced through the octree. For all visited cells the respective splats are checked for intersections with the ray until at least one splat is hit or the ray leaves the octree. If a ray intersects multiple splats within an octree cell, the intersection point with the smallest relative distance from the respective splat center is chosen, see Figure 4.4. For this intersection point the shading, reflection, and refraction model is applied, possibly using recursion, to compute the color.

For the intersection point a local normal is needed to apply the Blinn-Phong lighting model and to compute the directions of the reflected or refracted rays. Just using the normal field of the hit splat would necessarily lead to normal discontinuities between overlapping splats. To avoid the discontinuity, the normals of overlapping splats are averaged. Let $S_1, \ldots, S_p$ be all splats that are hit by a ray within a small environment around the intersection point $\mathbf{x}$. Moreover, let $(u_1, v_1), \ldots, (u_p, v_p)$ be the local coordinates of the ray intersection points and let $\mathbf{n}_1, \ldots, \mathbf{n}_p$ be the normals at these points obtained by the normal fields. Then the normal $\mathbf{n}$ at $\mathbf{x}$ is computed as

$$\mathbf{n} = \frac{\sum_{i=1}^{p} (1 - \|(u_i, v_i)\|_2) \cdot \mathbf{n}_i}{\sum_{i=1}^{p} (1 - \|(u_i, v_i)\|_2)} .$$

This averaging leads to continuously varying normals on the whole surface.

With this approach it is possible to produce high-quality and photorealistic images of surfaces in point-cloud representation. The linear normal fields on the splats and averaging between splats leads to smooth-looking surfaces although the splats only provide a piecewise linear surface representation even without C^0-continuity.

4.3 Image-space Point-cloud Rendering

A very different approach from the one presented in Section 4.2 will be explained now. Instead of generating photorealistic renderings with the help of expensive and complex computations, we propose an interactive yet smooth surface rendering without precomputations in object space. The approach goes back to the original ideas of Grossman and Dally [GD98] of directly rendering points instead of surface parts surrounding the points and carrying out all processing steps in image space.

The basic idea comes from the observation that the growing sizes of point clouds result in permanently growing point-per-pixel ratios for the rendered images. Consequently, sizes of rendering primitives go down to subpixel level and the use of points as rendering primitives gets feasible. However, also with high point-per-pixel ratios it might happen that some pixels of the rendered image exhibit incorrect information such as background color or occluded surface parts. These pixels should be corrected in image space. The pipeline of our proposed rendering technique is shown in Figure 4.5. It assumes that the point-per-pixel ratio is sufficiently high such that neighboring points on the surface are mapped to pixels that are not more than two pixels apart. This assumption is typically fulfilled when dealing with the densely sampled data sets we are commonly facing nowadays (up to a common zooming level).

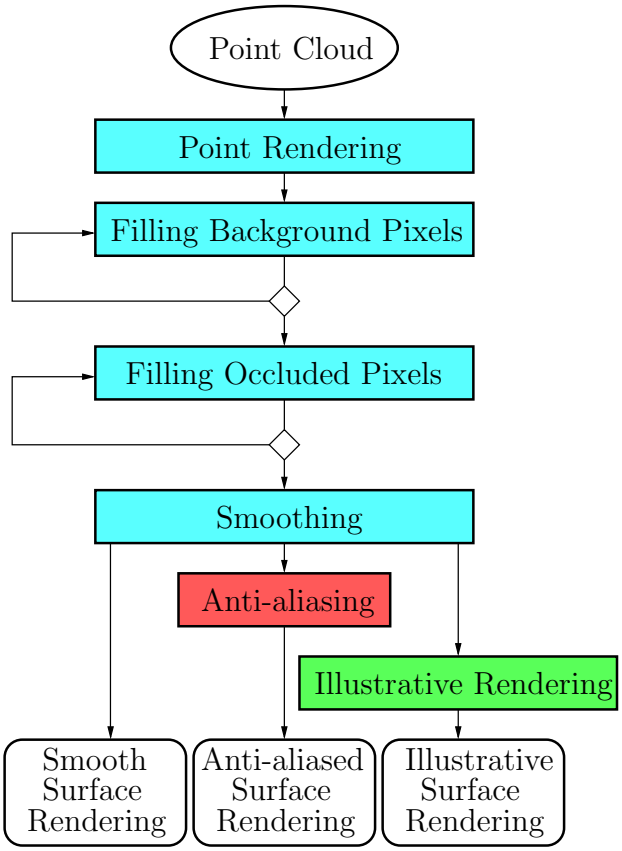


Figure 4.5: Process pipeline for image-space point-cloud rendering.

Starting from a point cloud including normals, the lit point cloud is rendered to a texture with color, depth, and normal information. This projection to image space is described in Section 4.3.1. Subsequently several filter operations are applied to image space.

In a second and a third step, possible holes in the projected surface are filled, see Section 4.3.2. These are holes that incorrectly exhibit background information or holes that exhibit occluded surface parts. The resulting piecewise constant surface representation does not exhibit holes anymore and is smoothed by a standard smoothing filter to generate smoothly varying color shades, see Section 4.3.3. This results in the desired smooth surface rendering.

Additionally the same steps can also be applied to the depth channel such that a subsequent edge detection filtering leads to a texture that exhibits the silhouettes and feature lines of the surface. This additional texture offers the opportunity of anti-aliasing the silhouettes, see Section 4.3.4, or performing illustrative rendering techniques like silhouette rendering, see Section 4.3.5.

Since all operations are performed in image space, they are implemented to operate on GPUs of today's graphics cards. Exploiting the capabilities of modern GPUs in terms of speed and parallelism, this allows the displaying of point clouds with large numbers of points at interactive rates and without any precomputations.

4.3.1 Point Rendering

All processing steps of the proposed rendering pipeline for point clouds are performed in image (or screen) space. Consequently, the first processing step is to project the point cloud $\tilde{\Gamma}$ into image space. Before projecting the points, they are lit using the local Blinn-Phong illumination model with ambient, diffuse, and specular lighting. The illuminated points are projected onto the screen using perspective projection.

During projection backface culling as well as depth buffering is applied. The backface culling is performed by discarding back-facing points $(\mathbf{x}_i, \mathbf{n}_i)$ with respect to the view point, i. e. those with $\langle \mathbf{n}_i, \mathbf{x}_v - \mathbf{x}_i \rangle < 0$, where $\mathbf{x}_v$ denotes the position of the viewer. These surface points can be discarded since they will not contribute to the final image.

The depth test is performed by using the OpenGL depth buffering. If two points are projected to the same pixel, the one closer to the viewer is considered. The colors of the projected points are stored in an RGBA color texture using the RGB channels only. Figure 4.6 shows the resulting texture for the skeleton hand data set consisting of 327,000 points.

Besides color and position in image space, the depth of each projected point, i. e. the distance of the represented point $\mathbf{x}_i$ to the viewer's position $\mathbf{x}_v$, is required for the subsequent pixel filling steps. However, the depth value calculated during the depth test cannot be used, since it is not linear in the distance d of the point to the viewer. Actually it is given by

$$f(d) := \frac{(d - z_{\text{near}})\, z_{\text{far}}}{(z_{\text{far}} - z_{\text{near}})\, d} ,$$

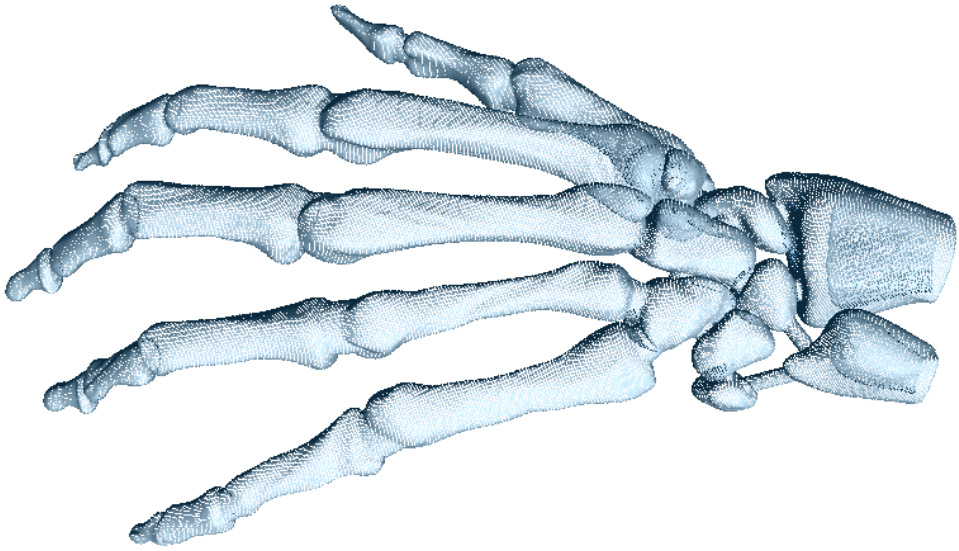


Figure 4.6: Rendering of the illuminated surface points of the skeleton hand data set containing 327k surface points. (Data set courtesy of Stereolithography Archive, Clemson University.)

where z_{near} and z_{far} denote the viewer's distances to the near and far planes.

However, depth values scaling linearly in the actual distances of the points to the viewer as well as lying in the range between 0 and 1 are needed to generate a consistent rendering with global depth thresholds. These linear depth values are calculated for each projected point by

$$f(d) := \frac{d}{z_{\text{far}}}$$

and stored at the respective position in the alpha channel of the RGBA color texture. Background pixels are stored with a depth value of 1.

4.3.2 Pixel Filling

If the sampling rate of surface Γ is high enough such that the projected distances of adjacent points of the point cloud $\tilde{\Gamma}$ are all smaller or equal to the pixel size, then the projected illuminated points that pass backface culling and depth test produce the desired result of a smoothly shaded surface rendering.

Obviously, this assumption does not hold in general. Especially when zooming closer to the object the resulting surface rendering exhibits "holes", such that pixels that should be filled with object colors are filled with background colors or colors of underlying surface layers, cf. Figure 4.6. Such pixels are filled in two consecutive steps with the proper surface color.

In a first pass, pixels incorrectly exhibiting background color need to be filled with the proper surface color and normal. However, which background pixels wrongly

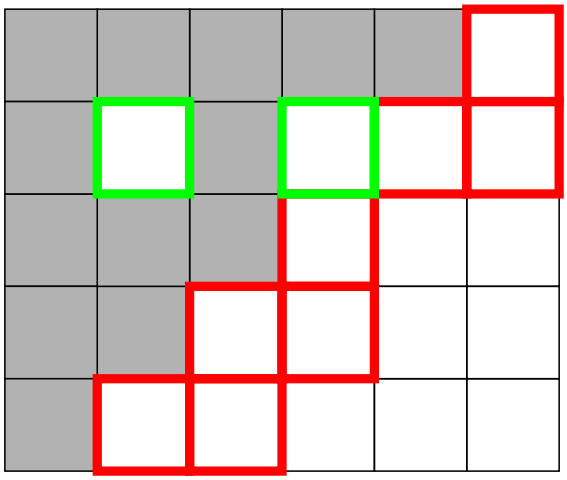


Figure 4.7: Closeup view of the rendered surface's border. Pixels that have been filled during point rendering are shown in grey, background pixels in white. All background pixels close to a filled pixel are framed. Green frames indicate holes in the surface rendering and the respective pixels have to be filled. Red frames indicate pixels that are beyond the silhouette of the object and must not be filled.

represent holes and have to be filled and which not has to be carefully chosen. Figure 4.7 shows the issue and the two cases that need to be distinguished. All white pixels represent background pixels. While the pixels with a green frame are examples of holes in the surface that result from low point density and need to be filled, the pixels with a red frame lie beyond the silhouette of the object and should maintain their background color.

The distinction between real background pixels and those which are to be filled is done by utilizing a filter on the image using 3×3 pixels. Obviously, this filter has to be applied only to those background pixels having non-background neighbors in their 3×3-pixels neighborhood. In Figure 4.7, the considered pixels are the ones that are framed. To identify these pixels that incorrectly exhibit background color (framed green in Figure 4.7) the eight masks shown in Figure 4.8 are used. Here white pixels indicate background pixels and the dark pixels could be both background or non-background pixels.

For each background pixel having at least one non-background neighbor, we test whether the 3×3 neighborhood of that pixel matches any of the cases. In case it does, the pixel is not filled. Otherwise it has to be filled. The new color, depth, and normal information is obtained from the pixel with the smallest depth out of the 3×3 neighborhood. This is a simple, yet very fast method which produces acceptable results.

The implementation of the test for background pixels is very simple and can be done efficiently by a single test. Assuming that background pixels have depth one, for each mask in Figure 4.8 the depth values of corresponding white pixels reduced by one are summed up. If the product of all eight sums equals zero, at

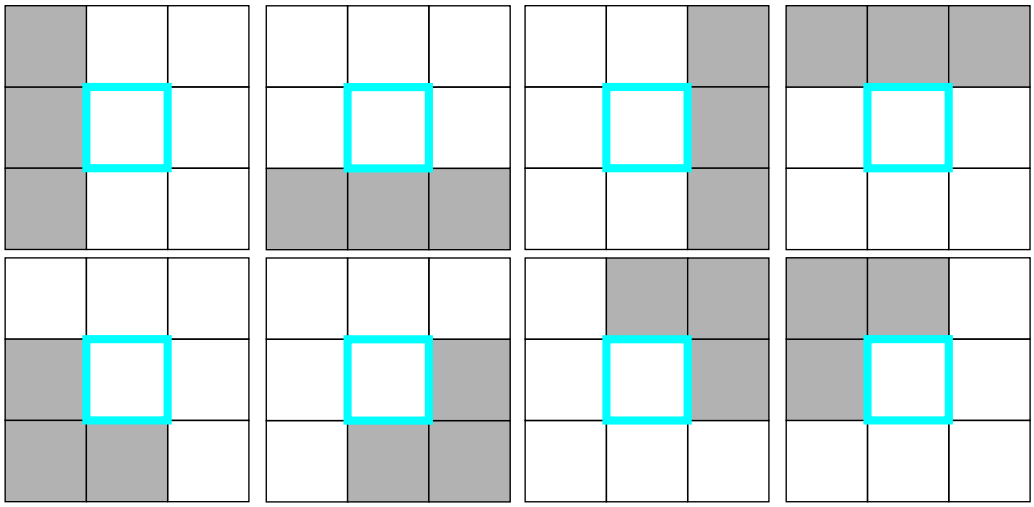


Figure 4.8: Filters with size 3×3 for detecting whether a background pixel is beyond the projected silhouette of the object. If one of the eight masks matches the neighborhood of a background fragment (blue), it is not filled. White cells indicate background pixels, dark cells may be background or non-background pixels.

least one sum was zero, i. e. at least one of the eight filters detected that the observed background pixel is beyond the object's silhouette.

The process of filling background pixels using the filter is iterated, until no more background pixels need to be filled. The number of iterations depends on the point density and the viewing parameters. It is easy to see that every hole in image space with a maximum diameter of n pixels is filled with at most n iterations. When applying the background filling to the output of the point rendering example, cf. Figure 4.6, the result shown in Figure 4.9 is produced with only one filter pass. The surface does not exhibit any holes with background color anymore. However, there might be still holes in the surface caused by pixels representing points of occluded surface parts. Such occluded pixels should also be replaced by color, normal, and depth values that represent the occluding surface part. This is achieved in an analogous way like the holes incorrectly exhibiting background color were filled.

Again, the pixels that represent parts of occluded surfaces have to be identified. In contrast to the considerations before, occluding and occluded pixels have to be distinguished by a threshold, i. e. a minimum distance $\tilde{d}$ between two consecutive surface layers. With the help of $\tilde{d}$ the border test is applied to all non-background pixels. For a candidate pixel with depth d, the used masks are similar to those in Figure 4.8, where, now, white pixels represent those pixels with depth values greater than $d + \tilde{d}$ and dark pixels may have any depth value. If the candidate pixel is identified as being occluded, its color, normal, and depth values are replaced by the values of the pixel with minimum depth within the filter's stencil.

In Figure 4.10, the effect of filling occluded pixels when applied to the skeleton hand

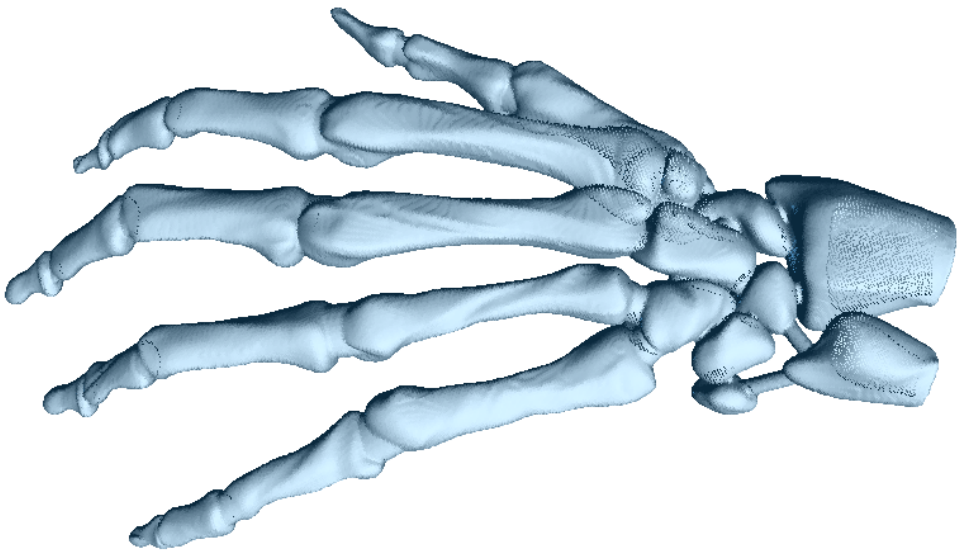


Figure 4.9: Filling background pixels applied to the output of point rendering (Figure 4.6) of the skeleton hand data set. Only one iteration of the filter had to be applied.

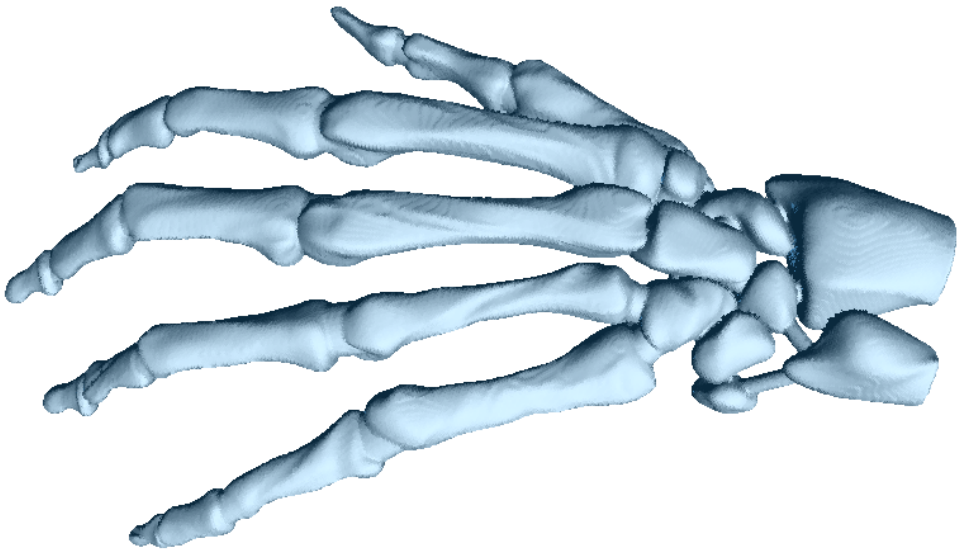


Figure 4.10: Skeleton hand data set after point rendering, filling background pixels, and one step of filling occluded pixels.

data set is shown. For the filling of occluded pixels, again only one iteration was needed. The application of the pixel filling steps to the projected lit point cloud results in a rendering of the surface exhibiting no holes anymore. However, the use of neighboring pixels of minimum depth for filling leads to a piecewise constant representation of the surface in image space.

4.3.3 Smoothing

In order to generate a smooth surface rendering from the piecewise constant image-space representation of the surface, image-based smoothing is applied to the output of the pixel filling step. For this smoothing a standard low-pass filter of size 3×3, such as the ones shown in Table 4.1 is applied to the image. Though both, the box filter and the Gaussian filter could be applied, we prefer an alleviated Gaussian filter, where the middle pixel is not weighted by 4 but by 16, to avoid blurring of the image. To avoid mixing of background and non-background colors, the smoothing filter is only applied to all non-background pixels, with adjusted weights respectively.

Box filter	Gaussian filter	alleviated Gaussian filter
$\frac{1}{9}\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}$	$\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}$	$\frac{1}{28}\begin{bmatrix}1&2&1\\2&16&2\\1&2&1\end{bmatrix}$

Table 4.1: Common low-pass filters of size 3×3. If a filter is applied to a pixel, it is assigned the weighted sum of the pixel itself and all neighboring non-background pixels with the given weights.

The smoothed version of Figure 4.10 is shown in Figure 4.11. A single iteration of applying the alleviated Gaussian filter suffices to produce the desired result of a smooth surface rendering. Additionally one can extend the pipeline by anti-aliasing or illustrative rendering techniques as explained in the following sections.

4.3.4 Anti-aliasing

When having a close-up look at the results generated by the image-space rendering, cf. Figure 4.12 (a), one can observe aliasing artifacts. The staircase effects become particularly obvious along the silhouettes and result from the strict classification of pixels in background and non-background in image space. Thus, it makes also no sense to apply the smoothing step to the whole image to remove the artifacts.

It rather makes sense to apply an additional anti-aliasing step to the silhouette pixels, i. e. pixels at the border of background pixels and non-background pixels as well as pixels at the border of a front surface layer and a back surface layer. To detect these silhouette pixels, a high-pass filter is applied to the respective depth values. Many high-pass filters exist and are commonly applied for edge detection.

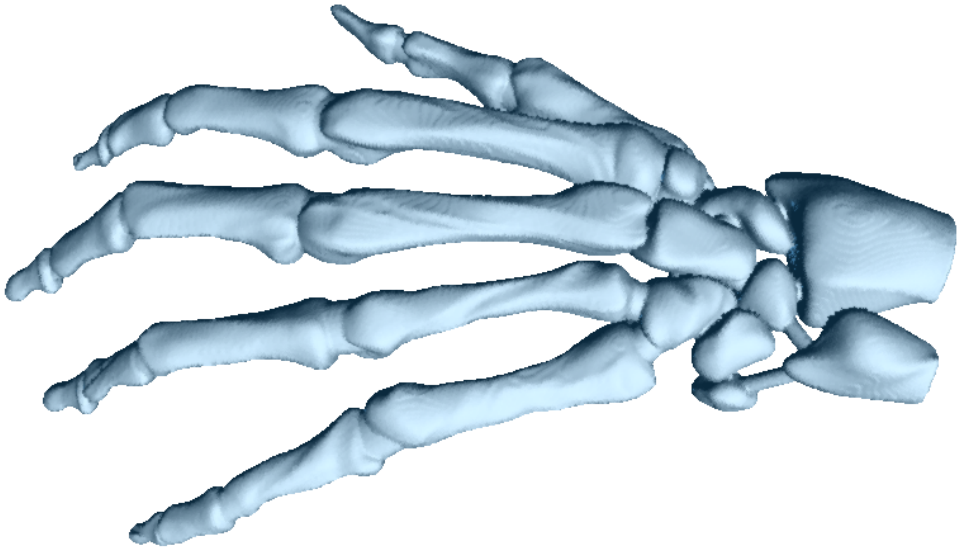


Figure 4.11: Point-cloud rendering of the skeleton hand data set after applying the entire image-space processing pipeline. For the final smoothing step, an alleviated Gaussian filter of size 3×3 has been applied once.

(a) (b)

Figure 4.12: (a) Close-up view on skeleton hand data set exhibits aliasing artifacts along the silhouette of the object. (b) Anti-aliasing by blending with high-pass-filtered depth buffer texture.

Any of these could be applied. We chose to apply a Laplace filter for our purposes, as one filter can simultaneously detect edges in all directions. Table 4.2 shows the Laplace filter of size 3×3 that was applied to our examples.

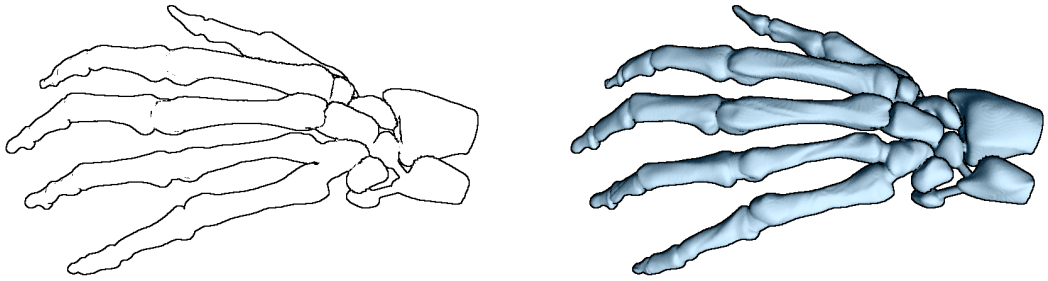
0	-1	0
-1	4	-1
0	-1	0

Table 4.2: Laplace filter used for edge detection.

Applying the Laplace filter to the depth values results in a texture with all silhouette pixels. This texture is blended with the color texture to obtain an anti-aliased image, cf. Figure 4.12 (b). Before blending the two textures, a thresholding can be applied to the high-pass-filtered depth information in order to decide whether only the background-foreground transitions should be anti-aliased (high threshold) or whether the front-back surface layer transitions should also be further anti-aliased (low threshold).

4.3.5 Illustrative Rendering

A second way of postprocessing the rendered images is to enhance depth perception and geometrical structure of the displayed surfaces by non-photorealistic rendering techniques. The possibility to produce illustrative drawings [ST90] is easily opened up by the detection of silhouettes described in the previous section and the concurrent processing of a normal map throughout the whole rendering pipeline.

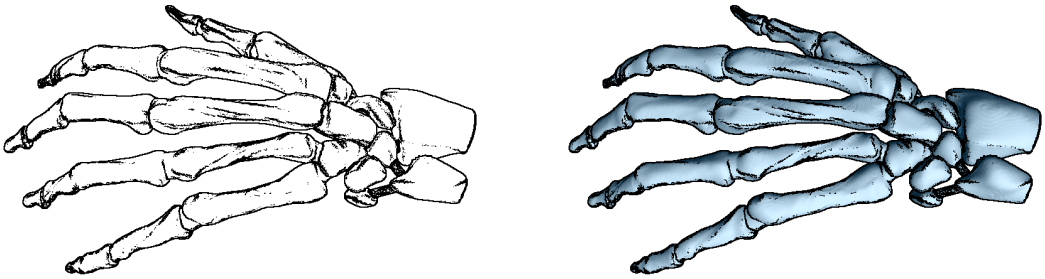


(a) (b)

Figure 4.13: (a) Silhouette rendering of skeleton hand data set. The silhouettes are detected by applying a 3×3 Laplace filter to the depth values. To make the silhouettes more visible, the lines have been thickened. (b) Combination of silhouette rendering with illuminated surface rendering of the skeleton hand data set.

Applying the Laplace filter of Table 4.2 to the depth values of the rendered image leads to a silhouette rendering. As described in the previous section, a thresholding can be used to adjust the amount of silhouettes that are rendered. Figure 4.13 (a) shows such a silhouette rendering for the skeleton hand data set, while

Figure 4.13 (b) shows the blending of silhouette rendering with the photorealistic rendering of Figure 4.11. Thus, the individual parts of the surface become much clearer and depth relations are more obvious.



(a) (b)

Figure 4.14: (a) Rendering of the feature lines of skeleton hand data set. The feature lines are detected by applying a 3×3 curvature filter to the normal map. (b) Final rendering of the illuminated skeleton hand data set, after applying the entire processing pipeline, smoothing and illustrative rendering.

The available normal map permits a yet much more illustrative type of rendering, by detecting regions with high surface curvature, i. e. feature lines. The surface curvature at a surface point can be easily obtained by exploring the dot product of surface normals at neighboring surface points [Fis89]. It is obtained in image space by observing surface normals of adjacent pixels. Hence, feature lines can be extracted from the final rendering, by applying a 3×3 filter on the normal map, highlighting regions with high curvature. An illustrative rendering of the feature lines of the skeleton hand data set is shown in Figure 4.14 (a). Figure 4.14 (b) shows the final rendering, including smooth surface rendering, silhouettes, and feature lines. Again a thresholding can be applied to the generated curvature map to control the accentuation of the surface features.

4.4 Results and Discussion

Both proposed rendering techniques for point clouds have been tested in terms of quality and computation speed. Therefore they were tested on a variety of different point-cloud data sets, either obtained by scanning surfaces of real objects, cf. [CL96], or isosurface extraction from unstructured point-based volume data, cf. Chapter 2. All experiments were performed on a single 2.66GHz Intel Xeon processor, supported by an Nvidia Quadro FX 4500 graphics card for the image-space rendering.

First the surface normal approximation technique was analyzed. The data independent computation times for different numbers of surface points and different numbers of nearest neighbors are given in Table 4.3. As expected the method based on kd-trees has a complexity of $O(n \log n)$ regarding the total number of surface points as well as regarding the number of nearest neighbors used for normal approximation.

Just for small numbers of surface points this behavior is not clearly visible since the computational overhead for generating the kd-tree is to high in relation to the actual computation of neighbors and normals.

# points	10 neighbors	20 neighbors	30 neighbors	40 neighbors
125k	4 sec	5 sec	7 sec	12 sec
250k	8 sec	11 sec	15 sec	24 sec
500k	15 sec	21 sec	31 sec	48 sec
1,000k	34 sec	46 sec	63 sec	99 sec

Table 4.3: Computation times for surface normal approximation. For different numbers of surface points and different numbers of nearest neighbors, respectively, the total times of surface normal computation are given in seconds. The computation times include the computation of nearest neighbors as well as the actual normal approximation.

The first step for splat-based ray tracing is the generation of the needed splats for the given point cloud. This requires the generation of a kd-tree for the surface points to obtain the nearest neighbors, the actual generation of the splats, as well as the calculation of the normal field for each splat. A comparison of the computation times for the splat generation for data sets with different number of surface points and diverse shapes is given in Table 4.4. Since the data sets have been rescaled to fit into the unit cube $[0, 1]^3$, we were able to use one global error threshold $\delta = 0.0001$ indicating the maximum orthogonal distance of covered points to the splat during the splat generation of all processed data sets. Note that the sphere surface is very smooth and exhibits no sharp features. Hence, the number of splats could be dramatically decreased in comparison to the number of surface points. Nevertheless, nearly all data sets exhibit smooth surface areas and the number of splats can be significantly reduced.

dataset	# points	# splats	time
fuel	34,665	28,379	4 sec
sphere	113,682	703	5 sec
skeleton hand	327,323	286,911	47 sec
Buddha	543,652	384,007	85 sec

Table 4.4: Computation times and number of generated splats for the splat generation step. The computation time includes the nearest neighbor calculation, splat generation, and computation of normal fields for each splat.

The actual computation times for the ray-tracing step heavily depend on the actual scene, i. e. the data set, the objects properties, and the ray-trace depth. Figure 4.15 shows a ray-traced image from a scene combining the skeleton hand and the sphere data set. The image was generated using ray-trace depth 4. The hand is reflective and the sphere exhibits the effects of reflection and refraction assuming that the

sphere is made of solid glass. To get an overview over the performance of the ray-tracing process in terms of computation speed, a comparison for different screen resolutions and ray-trace depths is given in Table 4.5. As expected the computation time for ray tracing increases linearly with the number of pixels. With respect to the ray-trace depth the time for ray tracing increases sublinearly, which mainly results from the minor chance for secondary rays to hit surfaces of the scene again.

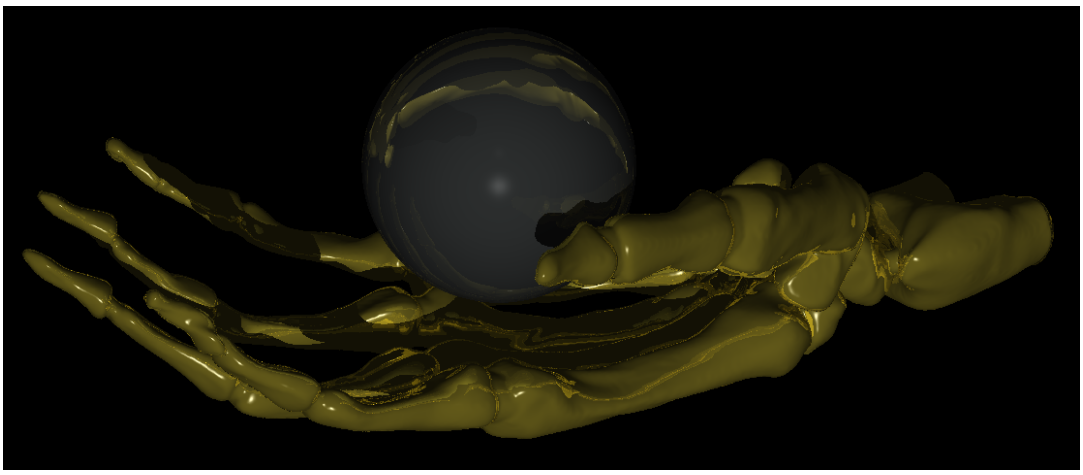


Figure 4.15: Splat-based ray tracing of the skeleton hand data set, combined with the sphere data set. For the rendering one white light source was used and the ray-trace depth was 4, leading to 115 seconds of computation time. The hand surface is reflective with a golden color. The sphere is made of solid glass, i. e. reflective and translucent, which leads to reflection effects of the hand in the lower hemisphere and refraction effects in the upper hemisphere of the sphere.

resolution	150^2	300^2	600^2	1200^2			
ray-trace depth	2	2	2	0	1	2	4
computation time	1 sec	5 sec	22 sec	45 sec	73 sec	88 sec	115 sec

Table 4.5: Comparison of computation times for ray tracing the scene from Figure 4.15 with different resolutions and ray-trace depths.

The performance of the splat-based ray-tracing approach in terms of quality, was analyzed with various additional data sets. A splat-based ray tracing of the scanned Buddha data set in front of a solid sphere is shown in Figure 4.16. The surface of the Buddha is golden and reflective. In contrast to Figure 4.15, the sphere is just reflective and colored blue. The scene includes three white light sources and was ray-traced with ray-trace depth 2. The rendering time for the image with 1200×1200 pixels was 408 seconds, which mainly results from the high number of splats, cf. Table 4.4, the few number of pure background pixels, and the complex geometry of the Buddha surface as indicated by the numerous self reflections on the surface.

As already shown in Chapters 2 and 3, the proposed ray-tracing technique is also able to produce high-quality images for point clouds obtained by direct surface extraction

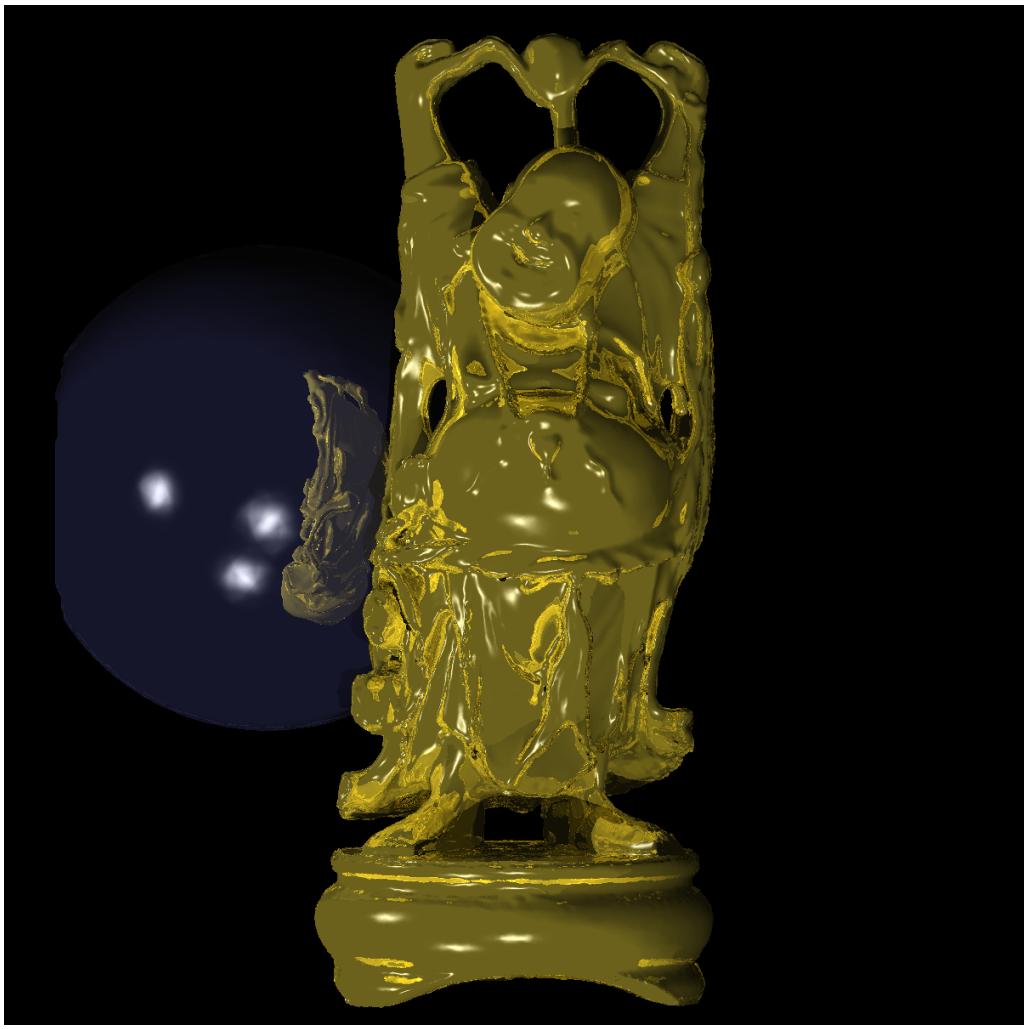


Figure 4.16: Splat-based ray tracing of the Buddha data set in front of a solid sphere rendered with ray-trace depth 2. Both surfaces are reflective and illuminated by three white light sources, leading to a rendering time of 408 seconds. The Buddha is colored golden while the sphere is dark blue. (Data set courtesy of Stanford Computer Graphics Laboratory.)

from unstructured point-based volume data. The fuel data set, cf. Table 4.4, was also created by direct surface extraction from unstructured point-based volume data. A ray tracing of the point cloud is shown in Figure 4.17. The scene consists of the reflective grey surface, illuminated by three different colored light sources. The computation time for rendering at a viewport of 1200×1200 pixels with a ray-trace depth of 2 was 84 seconds. In this picture, one can see the nice transitions of the light colors on the surface as well as the correct and realistic computation of partial shadows.

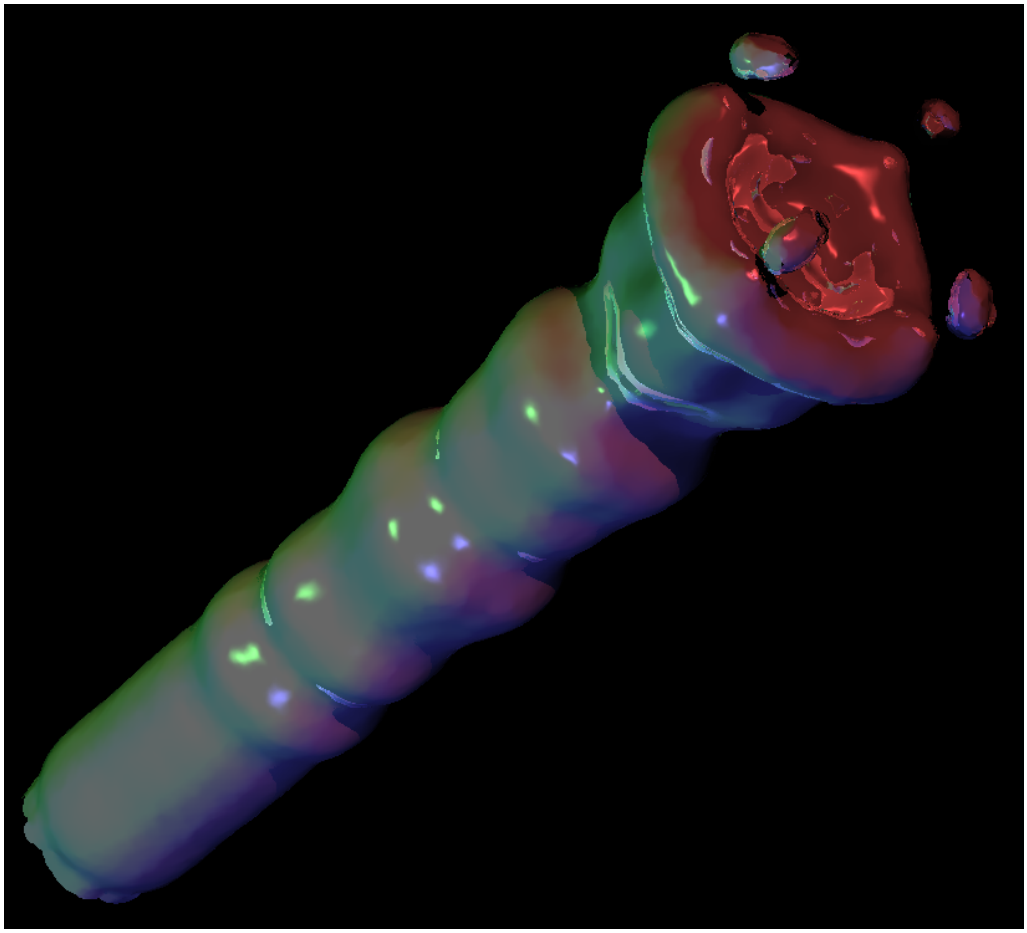
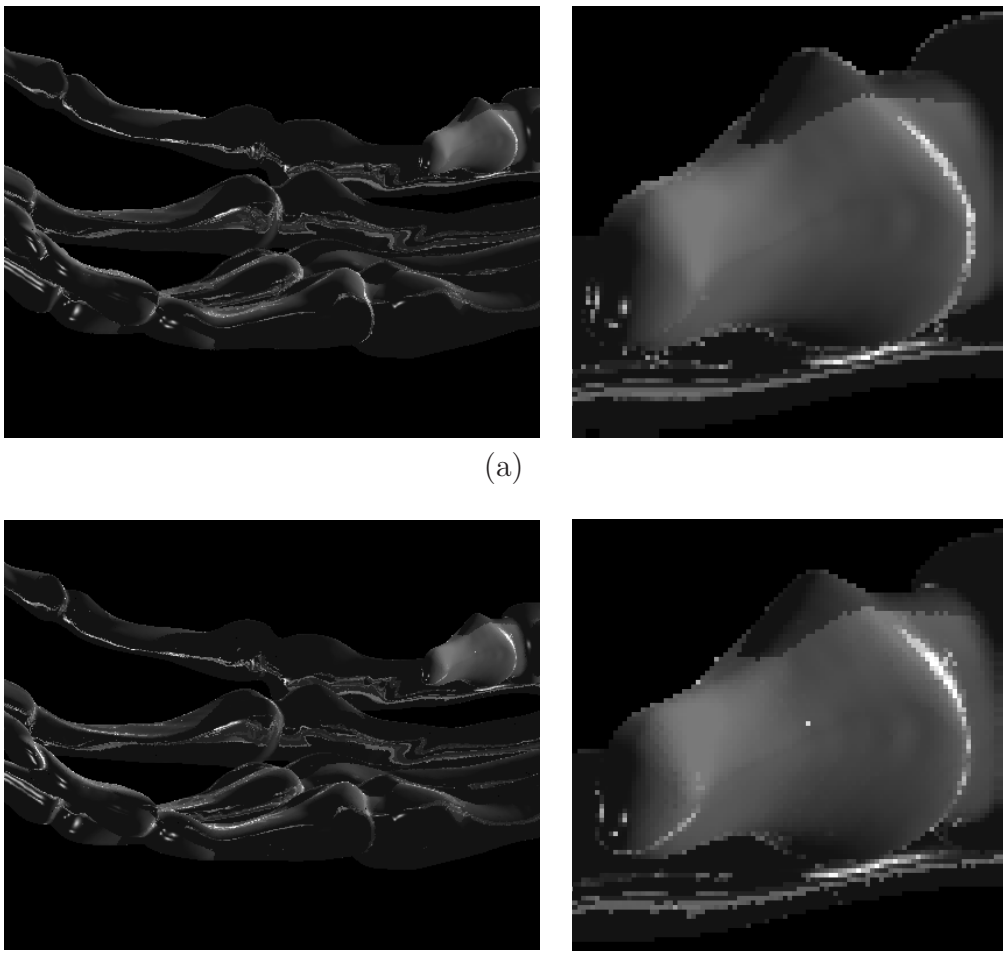


Figure 4.17: Splat-based ray tracing of the fuel data set obtained with ray-trace depth 2. The grey surface is reflective and illuminated by three light sources of different color (red, green, blue). The time for ray tracing was 84 seconds. (Data set courtesy of SFB 382 of the German Research Council (DFG).)

Altogether our presented splat-based ray-tracing method for point clouds produces very impressive renderings with photo-realistic effects and global illumination. Our computation times are clearly below the times we obtained with the approach of Wald and Seidel [WS05]. For the scene from Figure 4.18 and a ray-trace depth of 0, their approach lasted 120 seconds, while our approach only lasted 97 seconds. Comparing the achieved rendering quality, our splat-based ray tracing is at least equally

good. In fact, the method by Wald and Seidel exhibits some incorrectly shaded pixels which do not appear in our approach, as shown in Figure 4.18. Probably, these incorrect pixels could be circumvented by using different parameters in the challenged method, which would, however, lead to even higher computation times.



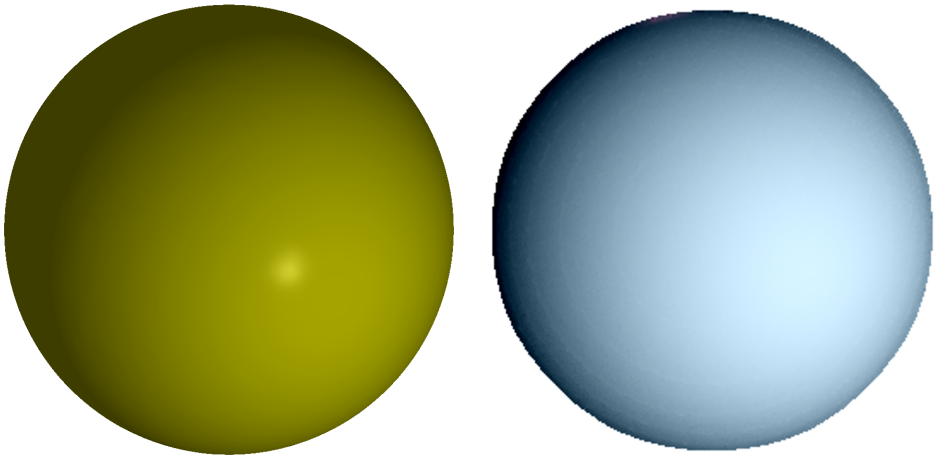
(a)

(b)

Figure 4.18: Comparison between splat-based ray tracing (a) and the ray-tracing technique proposed by Wald and Seidel (b). Both approaches were applied to the same scene of the hand data set with the same set of splats and a ray-trace depth of 2. The splat-based ray tracing lasted 128 seconds, while the challenged approach lasted 170 seconds and exhibits some incorrectly shaded pixels visible in the close-up view on the right.

The second proposed point-cloud rendering approach has a different purpose. Instead of using preprocessing time to compute a highly photorealistic but static picture as with splat-based ray tracing, the image-space point-cloud rendering produces high-quality and interactive renderings of point clouds without any precomputation. It is possible to directly explore smoothly shaded surfaces interactively generated based on the point clouds. A comparison between splat-based ray tracing and image-space

point-cloud rendering is shown in Figure 4.19. Both pictures, (a) and (b), show the same sphere data set rendered with the two proposed rendering techniques respectively. The quality of the ray-traced picture is, obviously, higher, since illumination and highlights are computed with the help of a normal field. Nevertheless, the surface generated by image-space point-cloud rendering is smooth and does not exhibit any holes. Additionally the rendering can be generated interactively in contrast to splat-based ray-tracing.



(a) (b)

Figure 4.19: Comparison of splat-based ray-tracing and image-space point-cloud rendering applied to the sphere data set respectively. In (a), a set of splats was generated from the data set and rendered using splat-based ray tracing with ray-trace depth 1. The overall computation time was 39 seconds. Note the very smooth surface although only 703 splats were computed to model the whole sphere. To the same data set with 113k surface points also the image-space point-cloud rendering technique was applied in (b). For filling the background pixels two filter steps have been applied followed by one smoothing step. Since the surface is convex, no filling of occluded pixels is needed. The average frame rate of the rendering was 34 fps.

The image-space rendering approach was also tested with the help of several point-cloud data sets. Again these have different origins. Some data sets were obtained by scanning 3D objects, like the skeleton hand data set used to visualize the method throughout Section 4.3. Additionally we considered the Buddha data set (courtesy of the Stanford University Computer Graphics Laboratory) and the dragon data set (courtesy of Stanford University Computer Graphics Laboratory). As data sets representing an isosurface of a volumetric scalar field we used the turbine blade data set (provided with the Visualization Toolkit) as well as the sphere data set.

The method was implemented in C++ with OpenGL. For the GPU implementation the OpenGL Shading Language was used. All experiments show that the proposed image-space rendering method is able to render point clouds with hundred thousands of points at interactive frame rates. A comprehensive overview over the computation

times for the presented data sets is given in Table 4.6. For each data set the respective computation times for point rendering, background pixel filling, occluded pixel filling, smoothing, and anti-aliasing or illustrative rendering is given with respect to two different viewport sizes. The actual frame rates for the overall rendering process and for the illustrative rendering technique are additionally given. Note that the frame rate is nearly halved when applying the illustrative rendering, since all pixel filling steps have to be applied to the normal field additionally.

data set (# points)	Dragon (437k)		Buddha (544k)	
viewport	512×512	1024×1024	512×512	1024×1024
point rendering	11.5 ms	12.0 ms	14.2 ms	14.4 ms
background fill. iter.	1.9 ms	8.6 ms	2.0 ms	9.3 ms
occluded fill. iter.	4.1 ms	14.7 ms	4.1 ms	15.8 ms
smoothing	0.9 ms	5.9 ms	1.0 ms	5.9 ms
anti-alias./illustrative	1.2 ms	1.9 ms	1.1 ms	1.9 ms
overall w. illustr. rend.	51 fps	20 fps	45 fps	18 fps
overall	**95 fps**	**38 fps**	**82 fps**	**35 fps**

data set (# points)	Sphere (113k)		Blade (883k)	
viewport	512×512	1024×1024	512×512	1024×1024
point rendering	7.0 ms	7.3 ms	24.8 ms	25.2 ms
background fill. iter.	1.9 ms	8.9 ms	1.1 ms	8.8 ms
occluded fill. iter.	4.0 ms	14.6 ms	3.8 ms	14.2 ms
smoothing	1.7 ms	7.9 ms	1.5 ms	7.0 ms
anti-alias./illustrative	1.1 ms	2.0 ms	1.1 ms	2.1 ms
overall w. illustr. rend.	44 fps	18 fps	31 fps	16 fps
overall	**80 fps**	**34 fps**	**60 fps**	**29 fps**

Table 4.6: Computation times for individual processing steps for three different data sets with two viewports. The time in milliseconds is given for each single computation step. The overall frame rate with illustrative rendering includes the complete processing pipeline applied to both, the color image and the normal map, with required number of iterations. In comparison, the overall frame rate for rendering the surface without illustrative rendering is given. All frame rates are given in frames per second (fps).

The very good performance of the image-space rendering technique in terms of quality is shown in the following pictures. All renderings have been generated using a 1024×1024 pixel viewport. Since the calculated depth values of pixels are linear and bounded to the interval $[0, 1]$ a global minimum depth threshold $\tilde{d} = 0.0001$ to distinguish between foreground and background, cf. Section 4.3.2, was used for all data sets.

Figure 4.20 shows an image-space rendering of the dragon data set consisting of 437,000 points. Background pixels have been filled using one filter step, while the filter for filling occluded pixels have been applied twice. Afterwards the image was

smoothed with one filter step. The average frame rate for the rendering without normal map was 38 frames per second.

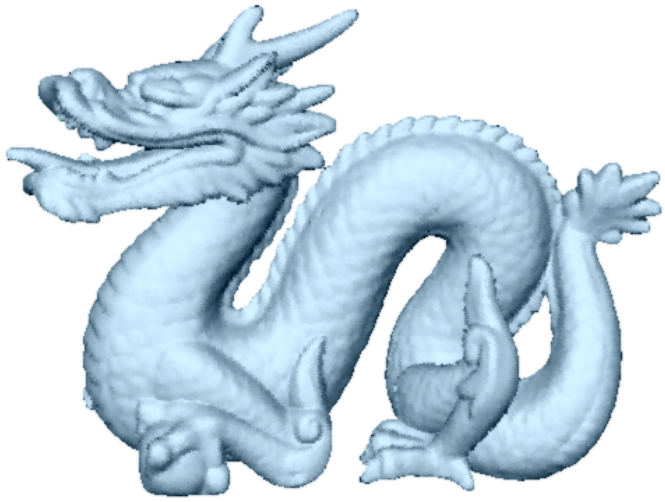


Figure 4.20: Image-space point-cloud rendering of the dragon data set with 437k surface points. Background pixels have been filled with one filter step, followed by two steps of the occluded pixel filling filter and one smoothing step. The average frame rate was 38 fps.

A picture showing the illustrative rendering technique to enhance depth perception is presented in Figure 4.21. For the Buddha data set with 544,000 surface points the filter pipeline was applied to the rendered points as well as to the normal map. Background pixels have been filled with one filter step, followed by two steps of the occluded pixel filling filter and one smoothing step. From the normal map feature lines have been extracted and blended with the rendered image to obtain the illustrative rendering. Even though the whole pipeline was applied to generate the illustrative rendering, it is still possible to obtain interactive frame rates of 18 frames per second on average.

Finally an anti-aliased rendering of the blade data set with 883,000 surface points is shown in Figure 4.22. Both background pixels and occluded pixels have been filled with three filter iterations, followed by one smoothing step. Subsequently anti-aliasing was applied to the silhouette of the surface. The average frame rate for the rendering was 29 frames per second. Note the perfect filling of occluded pixels even if consecutive layers of surfaces come close to each other.

Comparing our results to the pictures created by Grossman and Dally [GD98], shows a significant gain in rendering quality. In contrast to their renderings, our pictures are competitive with renderings of standard approaches applied to triangular meshes and other state-of-the-art point-cloud rendering algorithms, e. g. by Marroquim et al. [MKC07, MKC08]. Furthermore, the performance of our approach in terms of rendering times is better than the results presented by Marroquim et al. The authors state a frame rate of 21 frames per second for the Buddha data set and

Figure 4.21: Illustrative rendering of the Buddha data set with 544k surface points. Background pixels of the point rendering as well as the normal map have been filled with one filter step, followed by two occluded pixel filling steps and one smoothing step. Finally the feature lines have been extracted and blended with the rendering. The overall average frame rate of the rendering was 18 fps.

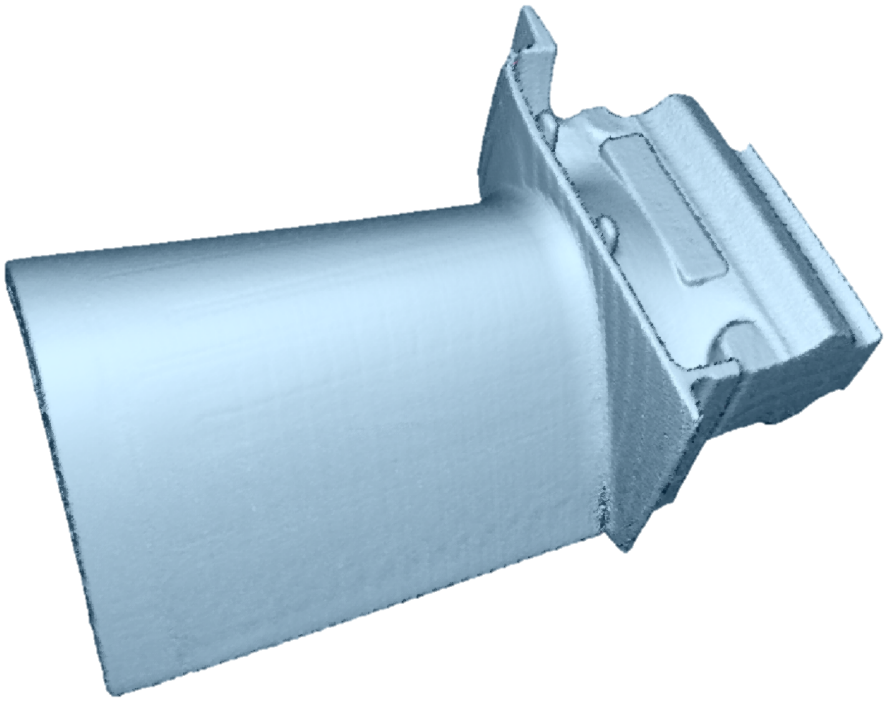


Figure 4.22: Anti-aliased image-space point-cloud rendering of the turbine blade data set with 883k sample points. Background pixels as well as occluded pixels have been filled with three filter iterations. Afterwards the image was smoothed with one filter step and anti-aliasing has been applied to the silhouette resulting in an average frame rate of 29 fps.

a resolution of 1024^2. Our method achieves a frame rate of 35 frames per second, rendering the same model and using comparable hardware.

Additionally and like most other point-cloud rendering algorithms, the method of Marroquim et al. does not allow the direct rendering of point clouds. Instead, a splat-like radius of influence is required for each surface point. The computation of such local surface reconstructions always induces the construction of space partitions and neighbor queries. Already for medium-sized data sets like the Buddha data set with 544k surface points this precomputation requires several seconds. In contrast, our approach is able to directly render point clouds without any precomputations, which is desired especially for the rendering of extracted isosurfaces or when visualizing time-varying point-cloud data sets.

However, the local image-space computations and ommission of any precomputation leads to the drawback that only actual holes can be filled. If the screen space exhibits only single pixels, each with an empty 2-neighborhood, which may result from very high zoom levels, the filters will not be able to detect the surface and fill the background. At these high zoom levels, the amount of visible points is just a small fraction of the original point cloud. A hybrid approach generating a local surface reconstruction could help in these cases to assure a consistent rendering quality [HTZL06, MKC08, WGK05].

In summary, our approach is capable of generating point-cloud renderings of high quality at interactive rates without any precomputations. For renderings with a high point-per-pixel ratio the results are competitive or even better than comparable methods in terms of rendering quality. In terms of rendering speed the achieved frame rates are better than state-of-the-art point-cloud rendering algorithms. In contrast to most competitive methods, the presented image-space point-cloud rendering technique can be directly applied to point clouds without any time-consuming precomputations.

Chapter 5

Conclusions

In this work we have proposed a visualization pipeline for surface extraction from unstructured point-based volume data. The pipeline includes all necessary steps to produce high-quality visualizations by extracting surfaces with different properties. An isosurface extraction step directly extracts isosurfaces in point-cloud representation. No global or local polyhedrization or reconstruction over a regular grid is applied. Instead, isopoints are computed by linearly interpolating between neighboring pairs of sample points. The neighbor information is retrieved by approximating natural neighbors as defined by Voronoi diagrams. We achieved surface extractions of high quality with significantly faster computation times than comparable approaches.

For noisy data or highly varying point densities, a level-set-based preprocessing step was introduced. This step generalizes the well-known technique of level sets, formerly only applicable to gridded data, such that it can be directly applied to unstructured point-based data without any reconstructions in data domain. By utilizing hyperbolic advection combined with mean curvature flow, the introduced level-set function is deformed such that its gradients stay normalized and the zero level set approximates the sought isosurface. By executing this step before isosurface extraction, we were able to extract smooth isosurfaces also from noisy or highly varying data. The achieved results are comparable in terms of quality with other approaches, that tackle similar problems. However, our level-set method is the only one able to be directly applied to unstructured point-based data.

As the surfaces are extracted in point-cloud representation, it is favorable to use point-cloud rendering techniques to visualize them. For this purpose, two different approaches have been presented. The first computes a set of circular splats with associated normal fields. Afterwards the set of splats is ray traced to generate a visualization of the surface which allows for photorealistic effects like global illumination, reflection, and refraction. In contrast, the second approach generates an interactive yet smooth visualization of the surface without any precomputations. The lit points are projected to screen space and possible holes are filled using image-space operations. Optionally the processing of an associated normal field allows the application of illustrative rendering techniques to enhance the depth perception of the interactive visualization. For both point-cloud rendering approaches we achieved results of

comparable visual quality and faster computation times, when compared to related state-of-the-art rendering techniques.

The presented visualization pipeline has been tested with the help of several data sets with different types of origin. The individual steps have been compared to competitive methods and assets and drawbacks have been discussed. The practicability of the pipeline is not only demonstrated on real-world data sets, but also directly in cooperation with data set providers [LLR09, LLRR08, RLL08, RRL07].

Bibliography

[AA03] Anders Adamson and Marc Alexa. Ray tracing point set surfaces. In *SMI '03: Proceedings of the Shape Modeling International 2003*, pages 272–279, Washington, DC, USA, 2003. IEEE Computer Society.

[AA04] Marc Alexa and Anders Adamson. On normals and projection operators for surfaces defined by point sets. In Marc Alexa, Markus Gross, Hanspeter Pfister, and Szymon Rusinkiewicz, editors, *Proceedings of the Eurographics Symposium on Point-based Graphics*, pages 149–156, Aire-la-Ville, Switzerland, 2004. Eurographics Association.

[ABCO+01] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin, and Claudio T. Silva. Point set surfaces. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 21–28, Washington, DC, USA, 2001. IEEE Computer Society.

[AGP+04] Marc Alexa, Markus Gross, Mark Pauly, Hanspeter Pfister, Marc Stamminger, and Matthias Zwicker. Point-based computer graphics. In *ACM SIGGRAPH 2004 Course Notes*, New York, NY, USA, 2004. ACM.

[AMM02] Sunil Arya, Theocharis Malamatos, and David M. Mount. Space-efficient approximate voronoi diagrams. In *STOC '02: Proceedings of the thirty-fourth annual ACM symposium on Theory of computing*, pages 721–730, New York, NY, USA, 2002. ACM.

[App68] Arthur Appel. Some techniques for shading machine rendering of solids. In *Proceedings of the AFIPS '68 Spring Joint Computer Conference*, volume 32, pages 37–45, New York, NY, USA, 1968. ACM.

[Aur91] Franz Aurenhammer. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1991.

[BCL06] Luc Buatois, Guillaume Caumon, and Bruno Lévy. GPU accelerated isosurface extraction on tetrahedral grids. In George Bebis, Richard Boyle, Bahram Parvin, Darko Koracin, Paolo Remagnino, Ara V. Nefian, Meenakshisundaram Gopi, Valerio Pascucci, Jiri Zara, Jose Molineros, Holger Theisel, and Thomas Malzbender, editors, *Advances in*

Visual Computing, Second International Symposium on Visual Computing, Proceedings, Part I, volume 4291 of *Lecture Notes in Computer Science*, pages 383–392, Berlin, Germany, 2006. Springer.

[BCMS08] Imma Boada, Narcis Coll, Narcis Madern, and J. Antoni Sellares. Approximations of 2D and 3D generalized Voronoi diagrams. *International Journal of Computer Mathematics*, 85(7):1003–1022, 2008.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[Ben80] Jon Louis Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980.

[Ben90] Jon Louis Bentley. K-d trees for semidynamic point sets. In *SCG '90: Proceedings of the sixth annual Symposium on Computational Geometry*, pages 187–197, New York, NY, USA, 1990. ACM.

[BF79] Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, 1979.

[Bil04] Stefan Bilbao. *Wave and scattering methods for numerical simulation*. John Wiley, Chichester, UK, 2004.

[Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. *ACM SIGGRAPH Computer Graphics*, 11(2):192–198, 1977.

[Bra00] Ronald N. Bracewell. *The Fourier Transform and Its Applications*. McGraw-Hill, New York, NY, USA, 3rd edition, 2000.

[BSK04] Mario Botsch, Michael Spernat, and Leif Kobbelt. Phong splatting. In Marc Alexa, Markus Gross, Hanspeter Pfister, and Szymon Rusinkiewicz, editors, *Proceedings of the Eurographics Symposium on Point-Based Graphics*, pages 25–32, Aire-la-Ville, Switzerland, 2004. Eurographics Association.

[BWMZ05] David Breen, Ross Whitaker, Ken Museth, and Leonid Zhukov. Level set segmentation of biological volume datasets. In Jasjit S. Suri, David L. Wilson, and Swamy Laxminarayan, editors, *Handbook of Biomedical Image Analysis, Volume I: Segmentation Models, Part A*, pages 415–478, New York, NY, USA, 2005. Kluwer.

[Car76] Manfredo Do Carmo. *Differential Geometry of Curves and Surfaces*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1976.

[CFL28] Richard Courant, Kurt Otto Friedrichs, and Hans Lewy. Über die partiellen Differenzengleichungen der mathematischen Physik. *Mathematische Annalen*, 100(1):32–74, 1928.

[CFL67] Richard Courant, Kurt Otto Friedrichs, and Hans Lewy. On the partial difference equations of mathematical physics. *IBM Journal of Research and Development*, 11(2):215–234, 1967.

[CGA] CGAL, Computational Geometry Algorithms Library. `http://www.cgal.org`.

[CHJ03] Christopher S. Co, Bernd Hamann, and Kenneth I. Joy. Iso-splatting: A point-based alternative to isosurface visualization. In Jon Rokne, Reinhard Klein, and Wenping Wang, editors, *Proceedings of the Eleventh Pacific Conference on Computer Graphics and Applications - Pacific Graphics 2003*, pages 325–334, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[CJ05] Christopher S. Co and Kenneth I. Joy. Isosurface generation for large-scale scattered data visualization. In Günther Greiner, Joachim Hornegger, Heinrich Niemann, and Marc Stamminger, editors, *Proceedings of Vision, Modeling, and Visualization 2005*, pages 233–240, Berlin, Germany, 2005. Akademische Verlagsgesellschaft Aka GmbH.

[CL96] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In John Fujii, editor, *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312, New York, NY, USA, 1996. ACM.

[Cle79] John Gerald Cleary. Analysis of an algorithm for finding nearest neighbors in euclidean space. *ACM Transactions on Mathematical Software*, 5(2):183–192, 1979.

[CPJ04] Christopher S. Co, Serban D. Porumbescu, and Kenneth I. Joy. Meshless isosurface generation from multiblock data. In Oliver Deussen, Charles D. Hansen, Daniel A. Keim, and Dietmar Saupe, editors, *Proceedings of the Eurographics/IEEE-TCVG Symposium on Visualization*, pages 273–282, Aire-la-Ville, Switzerland, 2004. Eurographics Association.

[DCH88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *SIGGRAPH Computer Graphics*, 22(4):65–74, 1988.

[Del34] Boris N. Delaunay. Sur la sphere vide. *Bulletin of Academy of Sciences of the USSR*, 7(6):793–800, 1934.

[DM72] Harry Dym and Henry P. McKean. *Fourier Series and Integrals*. Academic Press Inc., San Diego, CA, USA, 1972.

[Dun91] William Dunham. *Journey through Genius: The Great Theorems of Mathematics*. Penguin, New York, NY, USA, 1991.

[Ede87] Herbert Edelsbrunner. *Algorithms in combinatorial geometry*. Springer, Berlin, Germany, 1987.

[ELF04] Douglas Enright, Frank Losasso, and Ronald Fedkiw. A fast and accurate semi-lagrangian particle level set method. *Computers and Structures*, 83(6–7):479–490, 2004.

[EM94] Herbert Edelsbrunner and Ernst P. Mücke. Three-dimensional alpha shapes. *ACM Transactions on Graphics*, 13(1):43–72, 1994.

[ES91] Lawrence C. Evans and Joel Spruck. Motion of level sets by mean curvature I. *Journal of Differential Geometry*, 33(3):635–681, 1991.

[ES92a] Lawrence C. Evans and Joel Spruck. Motion of level sets by mean curvature II. *Transactions of the American Mathematical Society*, 330(1):321–332, 1992.

[ES92b] Lawrence C. Evans and Joel Spruck. Motion of level sets by mean curvature III. *Journal of Geometric Analysis*, 2(2):121–150, 1992.

[ES95] Lawrence C. Evans and Joel Spruck. Motion of level sets by mean curvature IV. *Journal of Geometric Analysis*, 5(1):77–114, 1995.

[ES98] Joachim Escher and Gieri Simonett. The volume preserving mean curvature flow near spheres. *Proceedings of the American Mathematical Society*, 126(9):2789–2796, 1998.

[FBF77] Jerome H. Freidman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977.

[Fis89] Robert B. Fisher. *From Surfaces to Objects: Computer Vision and Three Dimensional Scene Analysis*. John Wiley, New York, NY, USA, 1989.

[FN91] Richard Franke and Gregory M. Nielson. Scattered data interpolation: A tutorial and survey. In Hans Hagen and Dieter Roller, editors, *Geometric Modeling: Methods and Applications*, pages 131–160. Springer, New York, NY, USA, 1991.

[FP02] David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 2002.

[GD98] J.P. Grossman and William J. Dally. Point sample rendering. In *Rendering Techniques '98: Proceedings of the 9th Eurographics Workshop on Rendering*, pages 181–192, Wien, Austria, 1998. Springer.

[GF00] José Gomes and Olivier D. Faugeras. Reconciling distance functions and level sets. *Journal of Visual Communication and Image Representation*, 11(2):209–223, 2000.

[GH86] Michael E. Gage and Richard S. Hamilton. The heat equation shrinking convex plane curves. *Journal of Differential Geometry*, 23(1):69–96, 1986.

[GM77] Robert A. Gingold and Joe J. Monaghan. Smoothed particle hydrodynamics – theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181:375–389, 1977.

[Gra87] Matthew A. Grayson. The heat equation shrinks embedded plane curves to round points. *Journal of Differential Geometry*, 26(2):285–314, 1987.

[GST01] Sigal Gottlieb, Chi-Wang Shu, and Eitan Tadmor. Strong stability-preserving high-order time discretization methods. *SIAM Review*, 43(1):89–112, 2001.

[Hal05] Thomas C. Hales. A proof of the Kepler conjecture. *Annals of Mathematics*, 162(3):1063–1183, 2005.

[HCK+99] Kenneth E. Hoff III, Tim Culver, John Keyser, Ming Lin, and Dinesh Manocha. Fast computation of generalized Voronoi diagrams using graphics hardware. Technical report, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1999.

[HE03] Matthias Hopf and Thomas Ertl. Hierarchical splatting of scattered data. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, pages 433–440, Washington, DC, USA, 2003. IEEE Computer Society.

[HK05] Simone E. Hieber and Petros Koumoutsakos. A lagrangian particle level set method. *Journal of Computational Physics*, 210(1):342–367, 2005.

[HKL+99] Kenneth E. Hoff III, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 277–286, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[HLE04] Matthias Hopf, Michael Luttenberger, and Thomas Ertl. Hierarchical splatting of scattered 4d data. *IEEE Computer Graphics and Applications*, 24(4):64–72, 2004.

[HT05] Hsien-Hsi Hsieh and Wen-Kai Tai. A simple GPU-based approach for 3D Voronoi diagram construction and visualization. *Simulation Modelling Practice and Theory*, 13(8):681–692, 2005.

[HTZL06] Aimin Hao, Guifen Tian, Qinping Zhao, and Zhide Li. An accelerating rendering method of hybrid point and polygon for complex three-dimensional models. In Zhigeng Pan, Adrian Cheok, Michael Haller, Rynson W.H. Lau, Hideo Saito, and Ronghua Liang, editors, *Advances in Artificial Reality and Tele-Existence*, volume 4282 of *Lecture Notes in Computer Science*, pages 889–900, Berlin, Germany, 2006. Springer.

[JP00] Guang-Shan Jiang and Danping Peng. Weighted ENO schemes for Hamilton–Jacobi equations. *SIAM Journal on Scientific Computing*, 21(6):2126–2143, 2000.

[Kec98] Rainer Keck. Reinitialization for level set methods. Diploma thesis, University of Kaiserslautern, Germany, 1998.

[KKDH07] Michael Kazhdan, Allison Klein, Ketan Dalal, and Hugues Hoppe. Unconstrained isosurface extraction on arbitrary octrees. In *SGP '07: Proceedings of the fifth Eurographics symposium on Geometry processing*, pages 125–133, Aire-la-Ville, Switzerland, 2007. Eurographics Association.

[KN63] Shoshichi Kobayashi and Katsumi Nomizu. *Foundations of Differential Geometry, Volume 1*. John Wiley, New York, NY, USA, 1963.

[KN69] Shoshichi Kobayashi and Katsumi Nomizu. *Foundations of Differential Geometry, Volume 2*. John Wiley, New York, NY, USA, 1969.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, Redwood City, CA, USA, 1998.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM.

[Lev88] Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[LH95] Charles L. Lawson and Richard J. Hanson. *Solving Least Squares Problems*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1995.

[LLR09] Lars Linsen, Tran Van Long, and Paul Rosenthal. **Linking multi-dimensional feature space cluster visualization to surface extraction from multi-field volume data**. *IEEE Computer Graphics and Applications*, 29(3):85–89, 2009.

[LLRR08] Lars Linsen, Tran Van Long, Paul Rosenthal, and Stephan Rosswog. **Surface extraction from multi-field particle volume data using**

multi-dimensional cluster visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1483–1490, 2008.

[LMR07] Lars Linsen, Karsten Müller, and Paul Rosenthal. **Splat-based ray tracing of point clouds**. *Journal of WSCG*, 15:51–58, 2007.

[LP01] Lars Linsen and Hartmut Prautzsch. Global versus local triangulations. In Jonathan Roberts, editor, *Proceedings of Eurographics 2001, Short Presentations*, pages 257–263, Oxford, UK, 2001. Blackwell Publishers.

[LP03] Lars Linsen and Hartmut Prautzsch. Fan clouds - an alternative to meshes. In Tetsuo Asano, Reinhard Klette, and Christian Ronse, editors, *Geometry, Morphology, and Computational Imaging*, volume 2616 of *Lecture Notes in Computer Science*, pages 39–57. Springer, Berlin, Germany, 2003. Proceedings of the 11th International Workshop on Theoretical Foundations of Computer Vision.

[LR56] Peter D. Lax and Robert D. Richtmyer. Survey of the stability of linear finite difference equations. *Communications on Pure and Applied Mathematics*, 9:267–293, 1956.

[LT05] Stig Larsson and Vidar Thome. *Partial differential equations with numerical methods*. Springer, Berlin, Germany, 2005.

[Luc77] Leon B. Lucy. A numerical approach to the testing of the fission hypothesis. *Astronomical Journal*, 82(12):1013–1024, 1977.

[LXGF05] Chunming Li, Chenyang Xu, Changfeng Gui, and Martin D. Fox. Level set evolution without re-initialization: A new variational formulation. In *CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 1*, pages 430–436, Washington, DC, USA, 2005. IEEE Computer Society.

[MBNM07] Andreas Söderström Michael B. Nielsen, Ola Nilsson and Ken Museth. Out-of-core and compressed level set simulations. *ACM Transactions on Graphics*, 26(4):16, 2007.

[MBO94] Barry Merriman, James K. Bence, and Stanley J. Osher. Motion of multiple junctions: a level set approach. *Journal of Computational Physics*, 112(2):334–363, 1994.

[MBW+05] Ken Museth, David E. Breen, Ross T. Whitaker, Sean Mauch, and David Johnson. Algorithms for interactive editing of level set models. *Computer Graphics Forum*, 24(4):821–841, 2005.

[MBZW02] Ken Museth, David E. Breen, Leonid Zhukov, and Ross T. Whitaker. Level set segmentation from multiple non-uniform volume datasets. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 179–186, Washington, DC, USA, 2002. IEEE Computer Society.

[McN01] James McNames. A fast nearest-neighbor algorithm based on a principal axis search tree. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(9):964–976, 2001.

[MKC07] Ricardo Marroquim, Martin Kraus, and Paulo Roma Cavalcanti. Efficient point-based rendering using image reconstruction. In *Proceedings of the Symposium on Point-Based Graphics*, pages 101–108, 2007.

[MKC08] Ricardo Marroquim, Martin Kraus, and Paulo Roma Cavalcanti. Efficient image reconstruction for point-based and line-based rendering. *Computers & Graphics*, 32(2):189–203, 2008.

[MN03] Niloy J. Mitra and An Nguyen. Estimating surface normals in noisy point cloud data. In *SCG '03: Proceedings of the nineteenth annual Symposium on Computational Geometry*, pages 322–328, New York, NY, USA, 2003. ACM.

[MRH00] Arnold Meijster, Jos B.T.M. Roerdink, and Wim H. Hesselink. A general algorithm for computing distance transforms in linear time. In John Goutsias, Luc Vincent, and Dan S. Bloomberg, editors, *Mathematical Morphology and its Applications to Image and Signal Processing*, volume 18 of *Computational Imaging and Vision*, pages 331–340. Springer, Berlin, Germany, 2000. Proceedings of the fifth International Symposium on Mathematical Morphology and its Applications to Image and Signal Processing.

[Nie93] Gregory M. Nielson. Scattered data modeling. *IEEE Computer Graphics and Applications*, 13(1):60–70, 1993.

[OF03] Stanley Osher and Ronald Fedkiw. *Level set methods and dynamic implicit surfaces*. Springer, New York, NY, USA, 2003.

[OS88] Stanley. Osher and James A. Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics*, 79(1):12–49, 1988.

[OS91] Stanley Osher and Chi-Wang Shu. High order essentially non-oscillatory schemes for Hamilton-Jacobi equations. *SIAM Journal on Numerical Analysis*, 28(4):907–922, 1991.

[Pas04] Valerio Pascucci. Isosurface computation made simple: hardware acceleration, adaptive refinement and tetrahedral stripping. In Oliver Deussen, Charles D. Hansen, Daniel A. Keim, and Dietmar Saupe, editors, *Proceedings of the Eurographics/IEEE-TCVG Symposium on Visualization*, pages 293–300, Aire-la-Ville, Switzerland, 2004. Eurographics Association.

[PKG03] Mark Pauly, Richard Keiser, and Markus Gross. Multi-scale feature extraction on point-sampled surfaces. *Computer Graphics Forum*, 22(3):281–289, 2003.

[PLK+06] Sung W. Park, Lars Linsen, Oliver Kreylos, John D. Owens, and Bernd Hamann. Discrete Sibson interpolation. *IEEE Transactions on Visualization and Computer Graphics*, 12(2):243–253, 2006.

[PMO+99] Danping Peng, Barry Merriman, Stanley Osher, Hongkai Zhao, and Myungjoo Kang. A PDE-based fast local level set method. *Journal of Computational Physics*, 155(2):410–438, 1999.

[Pri07] Daniel J. Price. Splash: An interactive visualisation tool for smoothed particle hydrodynamics simulations. *Publications of the Astronomical Society of Australia*, 24(3):159–173, 2007.

[PTVF07] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3rd edition, 2007.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: surface elements as rendering primitives. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 335–342, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[RL00] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 343–352, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[RL06] Paul Rosenthal and Lars Linsen. **Direct isosurface extraction from scattered volume data**. In Beatriz Sousa Santos, Thomas Ertl, and Kenneth I. Joy, editors, *EuroVis06: Proceedings of the Eurographics/IEEE-VGTC Symposium on Visualization*, pages 99–106, Aire-la-Ville, Switzerland, 2006. Eurographics Association.

[RL08a] Paul Rosenthal and Lars Linsen. **Image-space point cloud rendering**. In *Proceedings of Computer Graphics International*, pages 136–143, 2008.

[RL08b] Paul Rosenthal and Lars Linsen. **Smooth surface extraction from unstructured point-based volume data using PDEs**. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1531–1546, 2008.

[RL09] Paul Rosenthal and Lars Linsen. **Enclosing surfaces for point clusters using 3D discrete Voronoi diagrams**. *Computer Graphics Forum*, 28(3):999–1006, 2009.

[RLL08] Paul Rosenthal, Tran Van Long, and Lars Linsen. **"Shadow Clustering": Surface extraction from non-equidistantly sampled multi-field 3D scalar data using multi-dimensional cluster visualization**. Winner of IEEE Visualization Design Contest, 2008.

[RO99] Alex De Robertis and Mark D. Ohman. A free-drifting mimic of vertically migrating zooplankton. *Journal of Plankton Research*, 21(10):1865–1875, 1999.

[RRHD08] Stephan Rosswog, Enrico Ramirez-Ruiz, William Raphael Hix, and Marius Dan. Simulating black hole white dwarf encounters. *Computer Physics Communications*, 179(1–3):184–189, 2008.

[RRL07] Paul Rosenthal, Stephan Rosswog, and Lars Linsen. **Direct surface extraction from smoothed particle hydrodynamics simulation data**. In Werner Benger, Ren Heinzl, and Wolfgang Kapferer, editors, *Proceedings of the 4th High-End Visualization Workshop*, pages 50–61, Berlin, Germany, 2007. Lehmanns Media.

[Sam06] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Elsevier, Amsterdam, The Netherlands, 2006.

[Set99] James A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, Cambridge, UK, 2nd edition, 1999.

[SF99] Mark Sussman and Emad Fatemi. An efficient, interface-preserving level set redistancing algorithm and its application to interfacial incompressible fluid flow. *SIAM Journal on Scientific Computing*, 20(4):1165–1191, 1999.

[SGM05] Avneesh Sud, Naga Govindaraju, and Dinesh Manocha. Interactive computation of discrete generalized Voronoi diagrams using range culling. In *ISVD '05: Proceedings of the 2nd International Symposium on Voronoi Diagrams in Science and Engineering*, 2005.

[Shu88] Chi-Wang Shu. Total-variation-diminishing time discretization. *SIAM Journal on Scientific and Statistical Computing*, 9(6):1073–1084, 1988.

[SHW05] Raul San Jose Estepar, Steve Haker, and Carl-Fredrik Westin. Riemannian mean curvature flow. In George Bebis, Richard Boyle, Darko Koracin, and Bahram Parvin, editors, *Advances in Visual Computing, First International Symposium on Visual Computing, Proceedings*, volume 3804 of *Lecture Notes in Computer Science*, pages 613–620, Berlin, Germany, 2005. Springer.

[SJ00] Gernot Schaufler and Henrik Wann Jensen. Ray tracing point sampled geometry. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 319–328, London, UK, 2000. Springer.

[SPG03] Christian Sigg, Ronald Peikert, and Markus Gross. Signed distance transform using graphics hardware. In *VIS '03: Proceedings of the 14th IEEE Visualization*, pages 83–90, Washington, DC, USA, 2003. IEEE Computer Society.

[SPL04] Miguel Sainz, Renato Pajarola, and Roberto Lario. Points reloaded: Point-based rendering revisited. In Marc Alexa, Markus Gross, Hanspeter Pfister, and Szymon Rusinkiewicz, editors, *Proceedings of the Eurographics Symposium on Point-based Graphics*, pages 121–128, Aire-la-Ville, Switzerland, 2004. Eurographics Association.

[SSS06] John Schreiner, Carlos E. Scheidegger, and Claudio T. Silva. High-quality extraction of isosurfaces from regular and irregular grids. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1205–1212, 2006.

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-D shapes. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 197–206, New York, NY, USA, 1990. ACM.

[Str89] John C. Strikwerda. *Finite difference schemes and partial differential equations*. Wadsworth, Belmont, CA, USA, 1989.

[Tho98] James W. Thomas. *Numerical Partial Differential Equations*. Springer, Berlin, Germany, 1998.

[TPG99] Graham M. Treece, Richard W. Prager, and Andrew H. Gee. Regularised marching tetrahedra: improved iso-surface extraction. *Computers and Graphics*, 23(4):583–598, 1999.

[TSE07] Eduardo Tejada, Tobias Schafhitzel, and Thomas Ertl. Hardware-accelerated point-based rendering of surfaces and volumes. In *Proceedings of WSCG 2007 Full Papers*, pages 41–48, 2007.

[TT97] Marek Teichmann and Seth Teller. Polygonal approximation of voronoi diagrams of a set of triangles in three dimensions. Technical report, Laboratory of Computer Science, MIT, 1997.

[Wat00] Alan Watt. *3D Computer Graphics*. Pearson, Essex, UK, 3rd edition, 2000.

[WGK05] Roland Wahl, Michael Guthe, and Reinhard Klein. Identifying planes in point-clouds for efficient hybrid rendering. In *Proceedings of the 13th Pacific Conference on Computer Graphics and Applications (Pacific Graphics 2005)*, 2005.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[WK04] Jianhua Wu and Leif Kobbelt. Optimized sub-sampling of point sets for surface splatting. *Computer Graphics Forum*, 23(3):643–652, 2004.

[WKL+03] Gunther H. Weber, Oliver Kreylos, Terry J. Ligocki, John M. Shalf, Hans Hagen, Bernd Hamann, and Kenneth I. Joy. Extraction of crack-free isosurfaces from adaptive mesh refinement data. In Gerald Farin, Hans Hagen, and Bernd Hamann, editors, *Approximation and Geometrical Methods for Scientific Visualization*, pages 19–40. Springer, Heidelberg, Germany, 2003.

[WLLP01] Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer S. J. Pister. Smart dust: Communicating with a cubic-millimeter computer. *Computer*, 34(1):44–51, 2001.

[WN08] Daniel Whalen and Michael L. Norman. Ionization front instabilities in primordial h ii regions. *The Astrophysical Journal*, 673:664675, 2008.

[WS03] Michael Wand and Wolfgang Straßer. Multi-resolution point-sample raytracing. In Torsten Möller and Colin Ware, editors, *Proceedings Graphics Interface 2003*, pages 139–148, Wellesley, MA, USA, 2003. A K Peters.

[WS05] Ingo Wald and Hans-Peter Seidel. Interactive ray tracing of point-based models. In *Proceedings of the Eurographics/IEEE VGTC Symposium Point-Based Graphics*, pages 9–16, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[ZPvBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 371–378, New York, NY, USA, 2001. ACM.

Declaration

I hereby declare that this thesis is my own work and effort and that it has not been submitted anywhere for any award. Where other sources of information have been used, they have been acknowledged.

_____

Date, Signature