Continuous reconstruction, rendering, and editing of point-sampled surfaces

Author(s):
Zwicker, Matthias

Publication Date:
2003

Permanent Link:
https://doi.org/10.3929/ethz-a-004617486

Rights / License:
In Copyright - Non-Commercial Use Permitted
Continuous Reconstruction, Rendering, and Editing of Point-Sampled Surfaces

A dissertation submitted to the
Swiss Federal Institute of Technology, ETH, Zurich

for the degree of
Doctor of Sciences

presented by
Matthias Zwicker
Dipl. Informatik–Ing. ETH
Swiss Federal Institute of Technology, ETH, Zurich
born March 18, 1973
citizen of Waldkirch, St. Gallen

accepted on the recommendation of
Prof. M. H. Gross, examiner
Prof. L. McMillan, co-examiner
Dr. H. Pfister, co-examiner

2003
The mathematical representation of surfaces is at the core of most computer graphics problems. In particular, the design of modeling and rendering algorithms for computer graphics applications is largely determined by the specific surface representation that is chosen. A variety of mathematical models has been developed for such applications, ranging from parametric to implicit surfaces. In this thesis, we propose to use nonuniform point samples to continuously reconstruct, render, and interactively edit surfaces. Based on results from classical signal processing theory, we develop core methods for reconstructing, filtering, and resampling nonuniformly point-sampled surfaces. From these concepts, we derive algorithms for high quality, antialiased rendering and interactive surface parameterization.

The term point sample denotes a sample of a surface, including information about the surface geometry and the surface appearance. A point-sampled surface consists of a set of point samples without connectivity information, where the points are distributed nonuniformly in space. Point-sampled surfaces are attractive for computer graphics applications because they are easily acquired using range sensing techniques and because they are efficient for modeling and rendering due to their simple structure.

This thesis starts by reviewing fundamental signal processing theory, paying particular attention to an analysis of the aliasing phenomenon. We then derive the concept of resampling filters as an efficient antialiasing strategy. Next, we shift our attention to nonuniformly sampled signals and discuss a local filtering approach to nonuniform sampling and reconstruction.

Extending these local filtering techniques and combining them with resampling filters, we introduce a parametric representation of point-sampled surfaces. We show how to interpret rendering as resampling a parameterized surface, and we derive an antialiased rendering algorithm for point-sampled surfaces. In addition, we present an adaptation of the algorithm suitable for hardware acceleration, and we present an efficient, hierarchical data structure tailored for point-based rendering. We also develop a resampling filter for volume data and design an efficient splatting algorithm for volume rendering.

Finally, we present a framework for interactive editing of point-sampled surfaces. We extend the concept of two-dimensional photo editing to three-dimensional objects by providing two fundamental operations on point-sampled surfaces, namely surface parameterization and surface resampling. Both these operators are based on the parametric representation of point-sampled surfaces. With these building blocks, we show how to implement editing operations such as painting, texturing, carving, displacement mapping, and filtering. Further, we explain a constrained minimum distortion parameterization algorithm that allows the user to perform sophisticated texturing operations. We also provide an efficient strategy for solving the resulting linear equation system using a nested iteration approach.
ZUSAMMENFASSUNG


ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor Prof. Markus Gross for giving me the opportunity to pursue my PhD studies in the Computer Graphics Laboratory at ETH Zürich. His enthusiasm has always been a great motivation and a driving force behind this project.

I would also like to thank Hanspeter Pfister from MERL, the Mitsubishi Electric Research Laboratories. Without his inspiration, this project would not have been started. As a co-advisor, he has contributed to this thesis with many helpful discussions and comments. In addition, my thanks go to MERL for accepting me repeatedly as a summer intern and for partially funding this project. It has always been a stimulating and enriching experience to work at MERL. Special thanks go to Jeroen van Baar and Liu Ren for helping with the design and the implementation of several algorithms developed in this project.

I would also like to express my gratitude to my co-advisor Leonard McMillan. This work has greatly benefited from his comments and advice.

Further, many thanks go to all my colleagues at the Computer Graphics Laboratory and the Perceptual Computing and Computer Vision Group. They have always created a very sociable and friendly atmosphere. Special thanks go to Mark Pauly for many groundbreaking and inspiring discussions about points in computer graphics.

Finally, my thanks go to all the diploma and semester students who contributed to this work: Christoph Niederberger, Alexandre Desboeufs, and Oliver Knoll. They all have helped me to finish this thesis with their enthusiasm and persistence in implementing many of the algorithms developed during this project.
CONTENTS

1 Introduction ................................................................. 1
  1.1 Point-Sampled Surface Representations .............................. 2
  1.2 Related Work .......................................................... 2
    1.2.1 Rendering ......................................................... 3
    1.2.2 Modeling, Animation and Simulation ............................ 3
    1.2.3 Continuous Surface Reconstruction ............................ 4
  1.3 Contribution ........................................................... 5
  1.4 Thesis Overview ....................................................... 6

2 Signal Processing and Aliasing in Computer Graphics .................. 9
  2.1 Signal Processing Fundamentals ...................................... 10
  2.2 Frequency Analysis of Aliasing ....................................... 11
  2.3 Aliasing in Computer Graphics ....................................... 12
  2.4 Antialiasing ............................................................ 16
    2.4.1 Prefiltering ...................................................... 16
    2.4.2 Postfiltering .................................................... 17
    2.4.3 Discussion ...................................................... 19
  2.5 Digital Filters For Computer Graphics .............................. 20
  2.6 Gaussian Filters ...................................................... 21
    2.6.1 One-Dimensional Gaussian Filters ............................. 21
    2.6.2 Multidimensional Gaussian Filters and Their Analytical Properties ... 23
    2.6.3 Two-Dimensional Gaussians for Image Processing .......... 25
  2.7 Resampling Filters .................................................... 26
    2.7.1 Destination Domain Resampling Filters ....................... 27
    2.7.2 Source Domain Resampling Filters ............................ 30
    2.7.3 Resampling Filter Properties ................................. 31
    2.7.4 Gaussian Resampling Filters .................................. 32

3 Nonuniform Sampling And Reconstruction ................................ 37
  3.1 Local Filtering ......................................................... 38
    3.1.1 Space Invariant Local Filtering ............................... 38
    3.1.2 Adaptive Local Filtering ..................................... 39
    3.1.3 Nonuniform Sampling .......................................... 43
# 4 Point Sampled 3D Geometry

4.1 Related Work

4.2 Smooth Surface Reconstruction from Parameterized Point Samples

4.3 Comparison to Moving Least Squares Approximation

# 5 EWA Surface Splatting

5.1 Continuous Surface Reconstruction in Image Space

5.2 Surface Resampling with Gaussian Filters

5.3 The Surface Splatting Algorithm

5.3.1 Splatting

5.3.2 Z-Buffering

5.3.3 Edge Antialiasing

5.4 Hardware Accelerated EWA Surface Splatting

5.4.1 Resampling Filter in the Local Parameterization Domain

5.4.2 Hardware Accelerated Rendering

5.5 Results and Comparison

5.5.1 Image Quality Comparison

5.5.2 Performance

# 6 Data Structures for Point-Based Rendering

6.1 The LDC Tree

6.1.1 Previous Work

6.1.2 LDC Tree Sampling

6.1.3 Octree Structure

6.1.4 Hierarchical Rendering

# 7 EWA Volume Splatting

7.1 Volume Resampling

7.1.1 Splatting Algorithms

7.1.2 The Volume Resampling Filter

7.1.3 The EWA Volume Resampling Filter

7.2 Implementation

7.3 Results and Comparison

# 8 Editing Parameterized Point-Sampled Surfaces

8.1 System Overview

8.2 Minimum Distortion Parameterization

8.2.1 Objective Function

8.2.2 Discrete formulation

8.2.3 Nested Iteration Least Squares Solver

8.2.4 Defining a Continuous, Smooth Surface
### FIGURES

<table>
<thead>
<tr>
<th>FIG.</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Frequency analysis of aliasing: aliasing free reconstruction</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Frequency analysis of aliasing: occurrence of aliasing</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Aliasing leads to the disintegration of a high frequency texture</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>Moiré patterns</td>
<td>15</td>
</tr>
<tr>
<td>2.5</td>
<td>Silhouette edges produce infinite frequencies in the output image</td>
<td>16</td>
</tr>
<tr>
<td>2.6</td>
<td>Filtering pipelines for digital image synthesis with antialiasing</td>
<td>18</td>
</tr>
<tr>
<td>2.7</td>
<td>Low-pass filter characterization</td>
<td>20</td>
</tr>
<tr>
<td>2.8</td>
<td>Gaussians in 1D</td>
<td>22</td>
</tr>
<tr>
<td>2.9</td>
<td>Flat field response of 1D Gaussians with varying standard deviation</td>
<td>23</td>
</tr>
<tr>
<td>2.10</td>
<td>Anisotropic 2D elliptical Gaussian</td>
<td>26</td>
</tr>
<tr>
<td>2.11</td>
<td>Image reconstruction on a regular sampling grid</td>
<td>27</td>
</tr>
<tr>
<td>2.12</td>
<td>Resampling consists of four steps</td>
<td>28</td>
</tr>
<tr>
<td>2.13</td>
<td>Source and destination space formulation of the Gaussian resampling filter</td>
<td>35</td>
</tr>
<tr>
<td>2.14</td>
<td>The behavior of Gaussian resampling filters in destination space</td>
<td>36</td>
</tr>
<tr>
<td>3.1</td>
<td>Ellipse coverage using Voronoi cells</td>
<td>41</td>
</tr>
<tr>
<td>3.2</td>
<td>Local Gaussian filters for a jittered grid</td>
<td>42</td>
</tr>
<tr>
<td>3.3</td>
<td>Gaussian filters computed using k-nearest neighbors</td>
<td>42</td>
</tr>
<tr>
<td>3.4</td>
<td>Comparison of sampling strategies</td>
<td>49</td>
</tr>
<tr>
<td>3.5</td>
<td>Visualization of the diagonal structure of the system matrices</td>
<td>50</td>
</tr>
<tr>
<td>3.6</td>
<td>Optimally sampled checkerboard texture</td>
<td>51</td>
</tr>
<tr>
<td>3.7</td>
<td>Comparison of sampling strategies with a high frequency texture</td>
<td>52</td>
</tr>
<tr>
<td>4.1</td>
<td>Curve reconstruction</td>
<td>59</td>
</tr>
<tr>
<td>4.2</td>
<td>Choosing appropriate reconstruction kernels</td>
<td>60</td>
</tr>
<tr>
<td>4.3</td>
<td>Curve smoothing</td>
<td>61</td>
</tr>
<tr>
<td>4.4</td>
<td>Surface smoothing</td>
<td>61</td>
</tr>
<tr>
<td>4.5</td>
<td>Comparison to MLS reconstruction</td>
<td>63</td>
</tr>
<tr>
<td>4.6</td>
<td>Reconstruction using our approach and MLS</td>
<td>63</td>
</tr>
<tr>
<td>5.1</td>
<td>Coordinate transformations</td>
<td>66</td>
</tr>
<tr>
<td>5.2</td>
<td>The mappings transform local parameters to image space coordinates</td>
<td>67</td>
</tr>
<tr>
<td>5.3</td>
<td>The surface splatting algorithm</td>
<td>69</td>
</tr>
<tr>
<td>5.4</td>
<td>Computing the bounding box for the resampling filter</td>
<td>71</td>
</tr>
<tr>
<td>5.5</td>
<td>Rasterizing the Gaussian resampling filter</td>
<td>71</td>
</tr>
<tr>
<td>5.6</td>
<td>Geometric object with intersecting, semi-transparent surfaces</td>
<td>73</td>
</tr>
<tr>
<td>5.7</td>
<td>Edge antialiasing</td>
<td>75</td>
</tr>
<tr>
<td>5.8</td>
<td>Mapping the unit Gaussian texture</td>
<td>76</td>
</tr>
<tr>
<td>5.9</td>
<td>Visibility splatting</td>
<td>78</td>
</tr>
<tr>
<td>5.10</td>
<td>Comparison of forward and backward mapping EWA filtering</td>
<td>80</td>
</tr>
<tr>
<td>5.11</td>
<td>Comparison of ellipse splatting and bilinear filtering</td>
<td>81</td>
</tr>
<tr>
<td>5.12</td>
<td>Comparison of elliptical Gaussian and bilinear reconstruction</td>
<td>81</td>
</tr>
<tr>
<td>5.13</td>
<td>Comparison of circle splatting and trilinear filtering (mip-mapping)</td>
<td>82</td>
</tr>
<tr>
<td>5.14</td>
<td>Zebra texture rendered with the different texture filters</td>
<td>83</td>
</tr>
</tbody>
</table>
FIG. 5.15: Checkerboard textures rendered using different texture filters. 84
FIG. 5.16: Test objects for the software implementation of EWA surface splatting. 86
FIG. 5.17: Test objects for the hardware accelerated surface splatting algorithm. 87
FIG. 6.1: LDC sampling. 93
FIG. 6.2: Analysis of LDC sampling. 94
FIG. 6.3: Eight-neighborhood of an LDI sample point. 95
FIG. 6.4: LDC sampling with and without redundant sample elimination. 96
FIG. 6.5: Uniformly subsampling an LDI to generate the nested subsets. 96
FIG. 6.6: On each level, the LDC tree is partitioned into blocks. 97
FIG. 6.7: The blocks are connected to a quadtree structure (2D example). 97
FIG. 6.8: Illustration of the extended sparse matrix data structure for storing LDIs. 98
FIG. 6.9: Pseudocode for accessing the sample points of an LDI. 98
FIG. 6.10: Pseudocode for hierarchical rendering using the LDC tree. 99
FIG. 6.11: Construction of the normal cone. 100
FIG. 6.12: Deriving the visibility cone from the normal cone. 101
FIG. 6.13: LOD Rendering. 103
FIG. 6.14: Rendering with different quality parameters. 103
FIG. 7.1: The forward mapping volume rendering pipeline. 106
FIG. 7.2: Volume rendering. 107
FIG. 7.3: Transforming the volume from camera to ray space. 112
FIG. 7.4: Mapping a reconstruction kernel from camera to ray space. 114
FIG. 7.5: Pseudo-code for the EWA volume splatting algorithm. 115
FIG. 7.6: Comparison between the EWA resampling filter and Swan et al. 116
FIG. 7.7: Comparison between EWA volume splatting and Swan et al. 116
FIG. 7.8: EWA volume splatting versus EWA surface splatting. 117
FIG. 7.9: Semi-transparent objects rendered using EWA volume splatting. 117
FIG. 7.10: Combining EWA volume and surface splatting. 118
FIG. 8.1: Overview of the framework for point-based surface editing. 121
FIG. 8.2: Minimum distortion parameterization. 123
FIG. 8.3: Computing directional derivatives using normal sections. 124
FIG. 8.4: Multilevel nested iteration scheme to solve Equation 8.16. 127
FIG. 8.5: Projection by parameterization. 128
FIG. 8.6: Resampling during interactive editing. 130
FIG. 8.7: Editing operations. 131
FIG. 8.8: Interactive texture mapping. 133
FIG. 8.9: Texture mapping with large distortions. 134
FIG. 8.10: Performance results of the nested iteration solver. 135
FIG. 8.11: Parameterizing a surface patch with resampling. 136
FIG. 8.12: Footprints on the moon. 136
TABLES

TABLE 3.1: Comparison of the error of the three sampling strategies. .......................... 50
TABLE 3.2: Comparison of the error of the three sampling strategies. ............... 52
TABLE 5.1: Rendering performance of the software implementation of EWA splatting. 86
TABLE 5.2: Rendering performance of hardware accelerated surface splatting. .... 87
TABLE 6.1: Rendering of the chameleon model with different quality parameters. 103
TABLE 8.1: Timings of the nested iteration solver. .................................................. 132
The mathematical representation of two-dimensional surfaces in three-dimensional space is at the core of many computer graphics problems. For example, algorithms for rendering surfaces to a two-dimensional image are a fundamental component of any computer graphics application. However, the design of these algorithms largely depends on the underlying surface representation. Further, to provide three-dimensional scenes and objects to be used in such applications, it is often necessary to refine raw data that was either acquired from real environments or that was manually generated. This involves modifying shape and appearance of the surfaces defining the three-dimensional objects.

There is a variety of surface representations that has been developed addressing the requirements of computer graphics applications, ranging from parametric surfaces (e.g., Bézier surfaces, NURBS, or subdivision surfaces) and triangle meshes to implicit surfaces (e.g., using radial basis functions, level sets, or sampled distance fields). In this thesis, we propose to use point samples as a fundamental building block to represent surfaces. By applying and extending results from classical signal processing theory, we develop core methods for reconstructing, filtering, and resampling nonuniform point-sampled surfaces. We also derive a novel parametric surface representation based on point samples.

Combining this surface representation with the concepts from signal processing theory, we devise high quality, antialiased rendering algorithms, which essentially render surfaces by performing a resampling operation. We also present a hierarchical data structure that is tailored for point-based rendering. Further, we extend the idea of rendering by resampling to volume data, leading to an efficient, antialiased volume rendering algorithm. Finally, we use the parametric surface representation in an interactive system for point-sampled surface editing.
1.1 POINT-SAMPLED SURFACE REPRESENTATIONS

In this thesis, we use the term *point sample* to denote a sample of a two-dimensional surface in three-dimensional space. A point sample includes information about the surface *geometry*, such as its position and its normal, and the surface *appearance*, i.e., its reflectance properties, but it does not have a spatial extent. Hence, surface representations based on point samples discretize surface geometry and surface appearance at the same rate. Essentially, a point-sampled surface consists of an arbitrary set of point samples without additional information. More specifically, the following two criteria are fundamental properties of point-sampled surface representations:

- **No connectivity:**
  There is no connectivity information between the point samples. In other words, each point is stored individually without explicit information about neighboring points on the surface. This is in contrast to, e.g., triangle meshes, where each triangle is defined by three vertices that are connected.

- **Nonuniform sampling:**
  The set of points defining the surface is distributed arbitrarily, or nonuniformly, in space; i.e., the sampling pattern is not restricted to uniform grids. In contrast, for example NURBS surfaces require a structured grid of control points, and subdivision surfaces require a semi-regular mesh.

Point-sampled surfaces are attractive for computer graphics applications for several reasons. Recent advances in range sensing techniques (laser range scanners, image-based methods, etc.) have enabled users to easily acquire complex three-dimensional models. The raw data produced by modern scanning devices are huge volumes of non-uniform point samples. A commonly used approach to process and render this data is generating triangle meshes or higher order surface representations (e.g., NURBS surfaces). However, to define these surfaces the connectivity between point samples has to be reconstructed, which is a difficult and time consuming process. In contrast, point-sampled surface representations are easily obtained from scanned data, since connectivity information is not stored in these representations. Further, it is often desirable to adapt the sampling pattern of a surface according to a modified geometry or appearance. For example, such *resampling* operations are performed during interactive surface modeling and editing or when multiresolution hierarchies are generated. Since the sampling patterns of point-sampled surfaces are neither restricted to uniform grids nor do the point samples store connectivity information, resampling is simple and efficient. Finally, point-sampled surfaces reduce the object representation to the essentials needed for rendering. In contrast, triangle primitives, which are used in most interactive rendering pipelines, implicitly store data that is not necessarily needed for rendering such as vertex valence or adjacency.

1.2 RELATED WORK

Point-based methods have been an active research area in computer graphics over the last two decades. Efforts have been made in investigating aspects as diverse as point-based rendering algorithms, using points as a primitive for modeling, animation, and simulation, and reconstructing continuous surfaces from point samples. In this section, we summarize fundamental contributions that are most related to our work.
1.2 RELATED WORK

1.2.1 Rendering

In fact, the concept of point-sampled surfaces has been introduced to computer graphics by Levoy and Whitted in the context of rendering [79]. In their pioneering report from 1985, they envisioned point-sampled surfaces as the *lingua franca* among the surface representations used in computer graphics. Only years later, points have been rediscovered as an efficient rendering primitive. The simplicity of point-based representations, avoiding the storage of any topology information, has been exploited by a number of researchers to develop rendering algorithms for highly complex surfaces.

Grossman and Dally’s work [46] was motivated by the desire to interactively render complex objects without the use of expensive hardware. They presented an efficient point-based rendering system that uses a hierarchical z-buffer to determine visibility and performs surface reconstruction in image space. Max [82] proposed to use points to represent the highly complex surfaces of trees. In his approach, points are stored in precomputed z-buffers and rendered using a hierarchical algorithm. The QSplat rendering system [114] was developed by Rusinkiewicz and Levoy in the course of the Digital Michelangelo Project [70] to quickly visualize the huge point clouds that were acquired by laser range scanners. Their rendering pipeline relies on a hierarchical data structure based on bounding spheres and on hardware accelerated point splatting. The randomized z-buffer algorithm introduced by Wand et al. [123] was designed to render scenes consisting of millions of objects at interactive rates. The algorithm is based on reconstructing images from a dynamically chosen set of random surface sample points. It facilitates the interactive display of scenes consisting of as many as \(10 \times 10^{14}\) triangles. It is interesting to note that all these approaches do not rely on a continuous surface defined by the points, but rather treat the points as discrete entities. As a consequence, none of those rendering algorithms includes a sound antialiasing scheme.

Closely related to the above point-based rendering techniques are approaches based on image warping. A comprehensive overview of the issues involved in image warping can be found in Wolberg’s book [142]. In particular, algorithms for warping depth images and layered depth images (LDIs) share many similarities with point-based rendering. In his work, McMillan focused on the analysis of visibility during depth image warping. He derived an ordering algorithm that allows to resolve visibility without a z-buffer [85, 86, 87], and he also extended this algorithm to cylindrical depth images [88]. A scalable hardware architecture for warping depth images was proposed by Popescu et al. [108]. This system was designed to allow the real-time display of natural scenes from arbitrary viewpoints. Oliveira presented a two-pass approach for warping depth images [99], which is based on a factorization into a one-dimensional pre-warp, followed by a planar projective mapping. Introduced by Shade et al. [123], layered depth images (LDIs) are an extension of depth images that store several depth values at each pixel location. With LDIs, it is possible to move the viewpoint and render disocclusion effects. The LDI tree [23] presented by Chang et al. combines the concept of LDIs with a space partitioning data structure. Hence, it allows for hierarchical rendering and progressive refinement. Lischinsky et al. [76] and Oliveira and Bishop [98] use a configuration of several LDIs to more uniformly sample object surfaces.

1.2.2 Modeling, Animation and Simulation

Similar to point samples, *particles* are nonuniformly distributed in space, do not store connectivity information, and do not have a spatial extent, but they store additional
attributes such as mass and velocity, and they respond to forces. Particle systems are widely used to simulate physical phenomena from the motion of stars in galaxies to electrons in semiconductor devices. A good introduction is the book by Hockney and Eastwood [52].

In computer graphics, particle systems are popular for the simulation of fluids, clouds, gases, fire, explosions, or smoke, because here visual realism is often more important than physical accuracy. A seminal contribution was Reeve’s work [111], where he introduced a general framework for modeling a class of fuzzy objects using particles. For the visual simulation of gaseous phenomena, Stam and Fiume solve the advection-diffusion equation using warped blobs [126]. Their method allows the visualization of intricate details of smoke and fire using only few particles (i.e., warped blobs). Enright et al. combine particles with level set methods [121] to simulate fluids [33], hence improving the accuracy of the simulations. Particles have also been used for surface modeling. Szeliski and Tonnesen [128] add an orientation to the state of each particle. They devise interaction potentials for their oriented particles that favor locally planar or spherical arrangements, which are suitable for surface modeling. Witkin and Heckbert [140] apply particles to sample and control implicit surfaces. They can make particles follow the surface for sampling, and the surface follow the particles for modeling. Finally, it has also been proposed to use point-sampled surfaces to represent animated objects. For example, Wand and Strasser [131] describe a multiresolution hierarchy consisting of points and triangles for key frame animation. Their method allows the representation of animated objects at different levels of detail.

1.2.3 Continuous Surface Reconstruction

There is an abundance of techniques for reconstructing continuous surfaces from unorganized points. However, most of these methods require the computation of connectivity information between the point samples, and their output is usually in the form of triangle meshes. Hoppe et al. [53] proposed a two-step procedure to reconstruct surfaces from unorganized points. In the first step, a signed distance function to the unknown surface is evaluated on a uniform grid. The surface is then extracted by contouring the scalar distance function (i.e., extracting an isosurface) using a variation of the marching cubes algorithm. Similar to [53], the method proposed by Curless and Levoy [25] proceeds in two steps, first computing a volumetric, scalar function, and then extracting an isosurface. However, their approach is limited to processing range images. This restriction is exploited when computing the scalar function, which is a weighted signed distance accounting for the properties of the scanning technology. Their approach also includes a technique for filling holes in the reconstructed surface, hence generating “watertight” meshes. Edelsbrunner and Mücke introduced the concept of $\alpha$-shapes [32]. Conceptually, $\alpha$-shapes are a generalization of the convex hull of a point set, where the parameter $\alpha$ controls the minimum size of cavities that appear in the shape (i.e., the $\alpha$-shape with $\alpha = \infty$ is the convex hull of the point set). The family of $\alpha$-shapes of a point set can be represented by its Delaunay triangulation. The crust algorithm by Amenta et al. [5] is also based on the Delaunay triangulation of the point set, but it does not require a user specified parameter $\alpha$ to define a shape. The authors provide a precise definition of a “good” sample of a surface and they prove that, given a good sample, their algorithm reconstructs a surface that is topologically equivalent to the input data.
While we do not know of any prior parametric, continuous surface definition based directly on nonuniform point samples without connectivity information used in computer graphics, there are several implicit representations that comply with our criteria for point-sampled surfaces. Point set surfaces [3] reconstruct a continuous, smooth surface using a so called moving least squares projection operator. Implicit surfaces based on nonuniform points have also been defined using radial basis functions (RBFs). In these approaches, an implicit function is represented as a weighted sum of basic functions, one basic function associated with each input point. The coefficients of the basic functions are then computed such that the resulting implicit function interpolates the input points. For an introduction to these methods see for example [115]. While it seemed to be intractable to solve problems with more than a few thousand points with RBFs until recently, Carr et al. [20] introduced fast methods that allow to model data sets with millions of points.

1.3 CONTRIBUTION

In this thesis, we take a signal processing approach to investigating point-sampled surfaces. Based on core methods for reconstructing, filtering, and resampling point-sampled surfaces, we devise antialiased rendering algorithms and a framework for interactive surface editing. The contributions of our work in this context can be summarized as follows:

- Destination space resampling filters
  We review the concept of resampling filters, which was introduced to computer graphics by Heckbert [49]. Resampling filters consist of a combination of a reconstruction and a low-pass filter. They are applied when a discrete signal is mapped from a source to a destination space; i.e., the reconstruction filter is used to generate a continuous signal in source space and the low-pass filter band-limits the signal in destination space before sampling. We derive a general formulation for destination space resampling filters, meaning that the reconstruction filter is transformed to destination space. We show that under the approximation that the mapping from source to destination space is affine, the destination space resampling filter can be expressed as the convolution of the reconstruction and the low-pass filter.

- Parametric point-sampled surface representation
  We introduce a parametric representation for point-sampled surfaces. With this representation, we are able to reconstruct continuous, smooth surfaces from nonuniform point samples without connectivity information. We exploit the versatility of this representation in the context of rendering and surface editing.

- Hierarchical data structure for efficient rendering
  We present a hierarchical data structure for efficient rendering of point-sampled surfaces. The data structure allows for level-of-detail and progressive rendering. It is storage efficient and, due to its semi-regular structure, allows to project the point samples to the image plane using an optimized image warping approach.

- Antialiased splatting algorithms
  Based on the destination space resampling filters, we design algorithms for anti-aliased rendering of point-sampled surface and volume data using a splatting
approach: For each point sample, we project a resampling filter into the output image. Using Gaussian reconstruction and low-pass filters, we derive explicit expressions for the destination space resampling filter for both surface and volume data.

- **Framework for interactive surface editing**
  Aiming at extending the functionality of two-dimensional photo editing systems to three-dimensional surfaces, we present a framework for interactive editing of point-sampled surfaces. We identify two key components of such a system, namely surface parameterization and surface resampling. We show how to implement a variety of surface editing operations, such as painting, texturing, carving, displacement mapping, and filtering, using parameterized point-sampled surfaces.

- **Constrained minimum distortion parameterization of point-sampled surfaces**
  We present an algorithm for constrained minimum distortion parameterization of point-sampled surfaces, which enables the user to perform sophisticated texturing operations in an interactive surface editing system. The algorithm is based on an objective function that simultaneously accounts for distortions and user specified constraints. This can be formulated as a least squares minimization problem leading to a system of linear equations, which we solve using a nested iteration approach.

### 1.4 THESIS OVERVIEW

This thesis is organized as follows:

- In Chapter 2, we first review fundamental concepts of signal processing theory, paying particular attention to a frequency analysis of the aliasing phenomenon. We summarize occurrences of aliasing effects in computer graphics and discuss different antialiasing strategies. After pointing out desirable characteristics of digital filters for computer graphics applications, we focus on Gaussian filters and explain their properties that will be exploited in this thesis. We then introduce the concept of resampling filters and show how to compute Gaussian resampling filters analytically.

- While signal processing theory deals with uniformly sampled signals, we shift our attention to nonuniformly sampled signals in Chapter 3. We discuss one particular approach to nonuniform sampling and reconstruction called local filtering, which is at the core of our surface reconstruction and filtering techniques.

- Chapter 4 introduces our parametric representation of point-sampled surfaces, extending the local filtering techniques of Chapter 3 from two-dimensional signals to surfaces in three-dimensional space. The surface definition is based on blending local surface approximations to a continuous, smooth surface by mapping the local approximations into a global parameter domain.

- In Chapter 5 we explain how to interpret rendering as resampling a parameterized surface, and we derive an antialiased rendering algorithm for point-sampled surfaces. The approach is based on destination space resampling filters using Gaussian reconstruction and low-pass filters. We show how to efficiently compute and evaluate the Gaussian resampling filters in destination space. We discuss an extended z-buffering approach that is used to determine visibility and that facili-
tates rendering semi-transparent surfaces and performing edge antialiasing. Further, we present an adaptation of the algorithm that is amenable to the implementation on current graphics hardware. The chapter is concluded by the presentation of qualitative and quantitative results: We illustrate the antialiasing capabilities and the high image quality of our approach compared to other filtering techniques. We also present performance numbers of the software and the hardware implementation of the rendering pipeline.

- Chapter 6 presents a hierarchical data structure tailored for point-based rendering. We review previous work in this context and point out how our data structure is related to these approaches. We then describe the sampling characteristics of the structure and discuss a general rendering algorithm exploiting its hierarchical nature.

- Chapter 7 describes our method for volume rendering using antialiased splatting. We formulate a destination space resampling filter for volume data based on Gaussian reconstruction and low-pass filters, similar as for point-sampled surfaces. This is achieved by analytically integrating three-dimensional Gaussian filters along viewing rays. Our approach is able to handle elliptical filters efficiently, hence it is suitable for rendering curvilinear and nonuniformly sampled volume data. We compare the results achieved with our resampling filter to other antialiasing techniques for volume rendering and provide performance data of our implementation.

- In Chapter 8 we present our framework for interactive editing of point-sampled surfaces. We extend the concept of two-dimensional photo editing to three-dimensional objects by providing two fundamental operations on point-sampled surfaces, namely surface parameterization and surface resampling. Both these operators are based on the surface representation introduced in Chapter 4. With these building blocks, we show how to implement editing operations such as painting, texturing, carving, displacement mapping, and filtering. Further, we explain a constrained minimum distortion parameterization algorithm that allows the user to perform sophisticated texturing operations. We also provide an efficient strategy for solving the resulting linear equation system using a nested iteration approach. Finally, we present results illustrating the implemented editing operations and documenting the performance of the minimum distortion parameterization algorithm.

- We conclude the thesis in Chapter 9 with a summary and directions for future work.

- The appendix includes a summary of the nomenclature used in Chapters 2 to 9, references, and color plates.
INTRODUCTION
Signal processing theory is an essential tool for dealing with the discrete nature of digital image synthesis. Specifically, it is instrumental in analyzing and understanding aliasing, which is a fundamental problem in computer graphics. Although conceptually computer graphics often deals with continuous representations of graphics models, in practice computer-generated images are represented by a discrete array of samples. Digital image synthesis [42] involves the conversion between continuous and discrete representations, which requires reconstructing and sampling multidimensional signals. This may cause aliasing artifacts that appear as visually disturbing effects in computer generated imagery. Aliasing artifacts include Moiré patterns and jagged edges, or flickering in animations.

Signal processing theory provides a set of tools precisely characterizing the relation between discrete, i.e., uniformly sampled, and continuous signals, where a discrete signal is a signal that is represented by individual signal values on a uniform grid. The main operations derived from this theory are the conversion of a continuous signal into a discrete signal (i.e., sampling), and the reconstruction of the continuous signal from the discrete signal. In particular, with the help of signal processing theory we can answer the following fundamental questions:

- Given a continuous signal and a uniform sampling grid, is it possible to first sample the signal and then reconstruct it from the samples, such that the reconstructed signal is identical to the original continuous signal? In other words, is there a sampled version of the signal on the given grid that is equivalent to the continuous signal?

- If this is not the case: Which is, in some meaningful way, the best sampled representation of the signal on the given grid, and how can we obtain it?
It turns out that the key to answering these questions is the conversion of continuous and discrete signals into the frequency domain using the Fourier transform. In this chapter, we review some definitions and results from signal processing theory and Fourier analysis, which are essential for analyzing the effects of sampling and understanding the relation between the continuous and the discrete representation of a signal.

2.1 SIGNAL PROCESSING FUNDAMENTALS

A filter is a process that takes a signal as an input and generates a modified signal or a response as an output. The easiest class of filters to understand are linear space invariant filters. Mathematically, a filter \( L \) is linear if

\[
L\{af + bg\} = aL\{f\} + bL\{g\}
\]

for any two scalars \( a \) and \( b \), and any two signals \( f : \mathbb{R} \to \mathbb{R} \) and \( g : \mathbb{R} \to \mathbb{R} \). It is space invariant if

\[
f(x) = L\{g(x)\} \iff f(x-s) = L\{g(x-s)\}
\]

for any spatial shift \( s \). A linear space invariant filter \( L \) is uniquely characterized by its impulse response \( h(x) \), i.e., its output resulting from an impulse input \( \delta \), where the impulse function \( \delta(x) \) is defined as

\[
\delta(x) = 0 \text{ if } x \neq 0 \text{ and } \int_{\mathbb{R}} \delta(x)dx = 1.
\]

It can be shown that, as a consequence [42], the response of a linear space invariant filter to any input signal \( f(x) \) is given by the convolution of \( f(x) \) and \( h(x) \):

\[
L\{f(x)\} = \int_{-\infty}^{\infty} f(t)h(x-t)dt = (f \otimes h)(x).
\]

Hence, convolution is as a linear operator on the signal, denoted by the symbol \( \otimes \).

A fundamental approach to analyze a filter is to compute its eigenfunctions and eigenvalues. The eigenfunctions of linear space invariant filters are complex exponentials, and the eigenvalues are given by the Fourier transform of its impulse response, which is called frequency response. The Fourier transform of a signal \( f(x) \) is called the spectrum of the signal, denoted by \( F(\omega) \). It is defined as

\[
F(\omega) = \int_{-\infty}^{\infty} f(x)e^{-j\omega x} dx,
\]

where \( \omega \) is the angular frequency. Likewise, the inverse Fourier transform describes the signal in terms of its spectrum:

\[
f(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega)e^{j\omega x} d\omega.
\]
2.2 FREQUENCY ANALYSIS OF ALIASING

We write \( f(x) \leftrightarrow F(\omega) \) to relate the spatial and the frequency domain representation of the signal. One of the most useful properties of the Fourier transform is that the Fourier transform of the convolution of two signals is the product of their Fourier transforms, i.e., \( f \otimes g \leftrightarrow FG \), and vice versa, i.e., \( fg \leftrightarrow F \otimes G/(2\pi) \) [14].

2.2 FREQUENCY ANALYSIS OF ALIASING

To study aliasing, it is useful to interpret images, surface textures, or volume data as multidimensional signals. In the following discussion, we will focus on one-dimensional signals to clarify the explanations, implying that the same concepts also hold in the multidimensional setting.

We analyze the sampling of a continuous signal using the Fourier transform and frequency domain representations, shown in Figure 2.1 and Figure 2.2. Sampling a continuous signal \( a_c(x) \) is performed by multiplying it with an impulse train \( i(x) \) (Figure 2.1b), which is a sum of unit-spaced impulses, i.e.,

\[
i(x) = \sum_n \delta(x - n).
\]

This yields the discrete signal \( a(x) = a_c(x)i(x/T) \), where \( T \) is the sample distance in the spatial domain. In the frequency domain, this results in the spectrum of the discrete signal \( A(\omega) \) given by the convolution \( A(\omega) = A_c(\omega) \otimes I(\omega)/(2\pi) \), where \( A_c(\omega) \) is the spectrum of the continuous signal \( a_c(x) \). The Fourier transform of the impulse train \( i(x/T) \) is another impulse train \( I(\omega) = \omega \delta(\omega/\omega_s) \), where the sample distance in the frequency domain \( \omega_s \) is inversely proportional to the sampling distance \( T \) in the spatial domain, i.e., \( \omega_s = 2\pi/T \). Since multiplication in the spatial domain corresponds to convolution in the frequency domain, the spectrum of the discrete signal consists of a superposition of replicas of the spectrum of the continuous signal spaced at a distance \( \omega_s \) (Figure 2.1c).

To reconstruct the spectrum of the continuous signal, we have to eliminate all replicas of \( A_c \) from \( A \) except the central one. This is achieved by multiplying \( A(\omega) \) with a box function \( H^{\omega_a}(\omega) = 1 \) for \( \omega \leq \omega_a \) and 0 otherwise, where \( \omega_a \) is the maximum frequency occurring in \( a_c \) or \( A_c \), respectively. \( H^{\omega_a} \) is called an ideal low-pass filter with cutoff frequency \( \omega_a \) (Figure 2.1d, right side), since it passes low frequencies \( \omega \leq \omega_a \) unchanged and suppresses high frequencies \( \omega > \omega_a \) completely. In the spatial domain, the impulse response of \( H^{\omega_a} \) is a sinc function:

\[
H^{\omega_a}(\omega) \leftrightarrow h^{\omega_a}(x) = 2\omega_a \frac{\sin(2\pi\omega_a x)}{2\pi\omega_a x}.
\]

Since the reconstructed spectrum \( A(\omega)H^{\omega_a}(\omega) \) in Figure 2.1e is identical to the original spectrum \( A(\omega) \) (Figure 2.1a), the inverse Fourier transform perfectly reconstructs the original signal (Figure 2.1e, left side).

However, if the maximum frequency \( \omega_a \) in the spectrum of \( A_c \) is higher than half the sampling distance in the frequency domain, i.e., \( \omega_a > \omega_s/2 \), the replicas overlap, as shown in Figure 2.2c. Hence it is impossible to reconstruct the original spectrum \( A_c \) from \( A \). When \( A \) is multiplied with the low-pass filter (Figure 2.2d), high frequencies
from the replicas appear as low frequencies in the original spectrum (Figure 2.2e), which is called **aliasing**: the high frequencies in the original signal masquerade, or alias, as low frequencies in the reconstructed signal (Figure 2.2e left side).

Aliasing need not occur if the continuous input signal is band-limited to a bandwidth $\omega_a$, i.e., it has no frequencies above $\omega_a$, or $A_s(\omega) = 0$ for $|\omega| \geq \omega_a$. A continuous signal with bandwidth $\omega_s$ can be reconstructed exactly if the sampling frequency $\omega_s$ is at least twice the bandwidth, i.e., $\omega_s > 2\omega_a$. This fact is known as the **Sampling Theorem**, and $\frac{\omega_s}{2}$ is called the **Nyquist frequency** of the signal. Equivalently, the frequency $\omega_s/2$ is also called the **Nyquist limit** of the sampling grid. Sampling a signal that contains frequencies above the Nyquist limit produces aliasing.

### 2.3 ALIASING IN COMPUTER GRAPHICS

Digital images are composed of a rectangular raster of individual picture elements, or pixels. Each pixel stores all image information, such as color, at a given point in the image. From a signal processing point of view, such raster images are discrete functions defined on a uniform 2D grid. Therefore, digital image synthesis is inherently a sampling process. The Nyquist limit of the 2D grid is determined by the distance between the sampling points, which is equivalent to the pixel size. According to the Sampling Theorem, an attempt to sample signals that contain frequencies above the Nyquist limit results in aliasing.

Aliasing appears in computer imagery in various forms as visually disturbing artifacts. We will classify aliasing artifacts into four categories: texture aliasing, edge aliasing, illumination aliasing, and temporal aliasing.

**Texture aliasing** refers to aliasing due to high frequency surface textures. Similar as in digital image warping [142], aliasing occurs if textures are not band-limited properly before they are sampled to the pixel grid of the output image. This leads to the disintegration of textures, illustrated in Figure 2.3, and so-called Moiré patterns, shown in Figure 2.4. Figure 2.3 shows a plane with a high frequency texture consisting of black text on a white background. In Figure 2.3a and 2.3b, most of the text is rendered unreadable due to aliasing of high frequencies. In contrast, the texture in Figure 2.3c and 2.3d was band-limited before sampling to obey the Nyquist limit of the pixel grid. This significantly improves the readability of the text. Similarly, the checkerboard texture in Figure 2.4a was not band-limited to the Nyquist limit of the sampling grid. As a consequence, Moiré patterns appear that clearly exhibit low frequencies not present in the input signal. Band-limiting the texture before sampling removes these artifacts, as shown in Figure 2.4b.

Projecting three dimensional scenes to an image plane leads to **silhouette edges** in the 2D image that contain infinite frequencies in general. Sampling edges on a raster image without precautions causes so-called **edge aliasing**, or “jaggies”. Jagged edges exhibit a staircase-like structure that is easily detected by the human visual system, as illustrated in Figure 2.5. Note that the color texture in this example is band-limited, leading to smooth color transitions on the object surface, which is in contrast to the abrupt transitions between foreground and background color at silhouette edges. Edge aliasing is even more objectionable in animations, where it is also referred to as “crawlies”, describing the crawling motion exhibited by moving edges.
FIGURE 2.1  Frequency analysis of aliasing: aliasing free reconstruction; (a) continuous input signal, (b) sampling grid, (c) sampled signal, (d) reconstruction filter, (e) reconstructed signal.
Frequency analysis of aliasing: occurrence of aliasing; (a) continuous input signal, (b) sampling grid, (c) sampled signal, (d) reconstruction filter, (e) reconstructed signal.

**FIGURE 2.2**
The evaluation of an illumination model is another source for high frequencies in computer generated images. Simulating light transport in a scene results in effects such as shadows, highlights, or reflections in the 2D image. These potentially contain infinite frequencies leading to artifacts similar to texture and edge aliasing, which we summarize as illumination aliasing.

Furthermore, creating animated image sequences does not only involve sampling on a two-dimensional pixel grid, but also a discretization in time. This additional sampling step is prone to aliasing artifacts, too, which are usually called temporal aliasing.
example, a spinning wagon wheel can appear to turn slowly backwards due to temporal aliasing. In this thesis, we restrict ourselves to deal with texture and edge aliasing, which lead to the most disturbing artifacts in interactive rendering applications.

2.4 **ANTIALIASING**

Let us define a digital image synthesis system as a mechanism that takes a specification of a continuous two-dimensional (2D) image signal \( g_c(x, y) \), as an input and turns it into a discrete image \( g(x_j) \) defined at discrete positions \( \{x_j\} \). The discrete image is then displayed by some physical device such as a monitor or a printer. Therefore, the discrete points \( \{x_j\} \) usually form a uniform square grid. Note that in general, the input function is not available as an explicit mathematical expression. Rather, the evaluation of \( g_c(x,y) \) requires a rendering algorithm, such as a scanline converter or a sophisticated ray tracing algorithm.

Hence, the image synthesis system essentially performs a sampling process. From the discussion in Section 2.2 we conclude that there are two fundamental approaches to reduce aliasing problems: either we eliminate frequencies above the Nyquist limit in the continuous signal before sampling, or we sample the signal at a higher frequency, i.e. increase the Nyquist limit of the sampling grid. The first approach is known as **prefiltering** in the literature, while the second alternative is called **postfiltering**.

2.4.1 **Prefiltering**

Figure 2.6a schematically illustrates digital image synthesis using prefiltering. The term prefiltering denotes the fact that the input signal is band-limited, i.e., filtered, before sampling, which is a theoretically sound procedure to avoid aliasing. Convolving the signal \( g_c(x) \) with an ideal low-pass filter \( h(x) \) yields a band-limited signal \( g'_c(x) = g_c(x) \otimes h(x) \) that respects the Nyquist limit of the sampling grid. Hence the
sampled image will be free of aliasing artifacts. Unfortunately, the input signal \( g_c(x) \) is in general not available in a form that allows the evaluation of this convolution. Remember that evaluating \( g_c(x) \) at an image point \( x \) is equivalent to computing the image color at \( x \) using a rendering algorithm. Prefiltering is often applied in the context of texture mapping [21, 13, 35, 49, 84, 54]. However, it is not applicable to avoid illumination and edge antialiasing.

### 2.4.2 Postfiltering

Figure 2.6b depicts the data flow in a postfiltering pipeline, which is a more practical approach to antialiasing. Conceptually, postfiltering starts with a continuous image function as an input, too. However, postfiltering does not rely on an explicit expression for \( g_c(x) \). Instead, an intermediate set of samples \( \{g_c(\xi_i)\} \) at positions \( \{\xi_i\} \) is acquired first. The objective of this step is to use a sampling grid \( \{\xi_i\} \) that introduces less aliasing than the sampling grid \( \{x_i\} \) of the output image. Next, a continuous signal \( \gamma_c(x) \) is reconstructed from this set of samples. Finally, this intermediate signal is band-limited to the Nyquist limit of the output sampling grid, yielding \( \gamma'_c(x) \), before \( \gamma'_c(x) \) is discretized to generate the final image samples at positions \( \{x_i\} \).

The effectiveness of postfiltering methods essentially depends on a clever choice of the sampling positions \( \{\xi_i\} \). On the one hand, the time complexity of most rendering algorithms is proportional to the number of image samples, i.e., \( |\{\xi_i\}| \), that are computed. On the other hand, using more sampling positions increases the Nyquist limit of the sampling grid and thus reduces aliasing. Much research has been devoted to finding configurations of sampling positions that optimize this trade-off between efficiency and antialiasing performance. In the literature [42], the various techniques are classified into two major categories:

**Uniform Sampling.** A straightforward approach to alleviate aliasing artifacts is to choose a uniform, or regular, intermediate sampling grid \( \{\xi_i\} \) that has a higher Nyquist limit than the output grid \( \{x_i\} \). Suppose that the Nyquist limit of \( \{x_i\} \) is \( a \) and the one of \( \{\xi_i\} \) is \( b \), \( a < b \). Hence supersampling removes aliasing of frequencies \( \omega \) in the band \( a < \omega < b \), while aliasing due to frequencies \( \omega > b \) persists.

Sampling on regular square grids is simple and computationally efficient, hence it is also suitable for hardware implementation. A series of hardware systems have been built that use brute force regular supersampling (e.g., [110]), or a variation of it called multisampling ([2, 47, 92]). Although it has been shown that regular hexagonal grids are more efficient than square grids for sampling two dimensional signals [29], such grids are rarely used in practice.

**Nonuniform Sampling.** From the frequency analysis of aliasing in Section 2.2 it becomes clear that the spectrum of the sampling grid determines the characteristics of the aliasing artifacts. Concentrated spikes of energy in the spectra of regular sampling grids replicate structures of the input signal in different frequency bands, leading to structured artifacts. Yellott [145] proposed two criteria for sampling grids to produce the least-conspicuous form of aliasing: first, their spectra should lack any spikes of energy, and second, they should have a deficiency of low-frequency energy. These criteria are known as the blue-noise criteria; blue-noise sampling grids have the ability to replace structured aliases by high-frequency noise, which is less objectionable to the human visual system [45]. Blue-noise sampling grids are inherently nonuniform, and there is a variety of methods to design such grids.
Filtering pipelines for digital image synthesis with antialiasing: (a) prefiltering, (b) postfiltering pipeline.
Based on the theory for analyzing stochastic impulse processes developed by Beutler and Leneman [8, 65, 9], stochastic sampling grids with blue-noise characteristics have been introduced to computer graphics [26]. The most popular sampling patterns are jittered grids and the Poisson disk distribution, which were analyzed and discussed by many authors, e.g., [26, 24, 90]. A Poisson disk distribution is a uniform random pattern that maintains a minimum distance criterion. Since it is computationally expensive to generate exact Poisson disk distributions, the so-called dart throwing algorithm is often used in practice to compute an approximation [24, 90, 83, 51].

It turns out that besides the spectral properties of a sampling pattern its so called discrepancy is another significant indication of its quality for image synthesis applications [124]. Intuitively, the discrepancy measures how well the samples are distributed over a domain. In particular, it has been proven mathematically that discrepancy is related to the approximation error in numerical integration (the Koksma-Hlawka inequality for integration, see also [50, 27]). Hence low discrepancy patterns are particularly interesting for Monte Carlo methods in light transport (i.e., global illumination) simulations. In addition to the stochastic sampling distributions mentioned above, various deterministic low-discrepancy sequences such as the Hammersley or Halton sequences have been introduced to computer graphics recently [50]. Even more efficient sampling grids, both in terms of approximation error and antialiasing characteristics, have been developed by randomizing deterministic low-discrepancy patterns [58].

Furthermore, it has been recognized early on that for efficient high quality image synthesis, the sampling rate has to be adapted locally to the image characteristics [137]. This led to the development of adaptive sampling techniques that comprise two main components: a refinement criterion and a refinement strategy. Adaptive sampling proceeds as follows: First, a set of initial samples is analyzed using the refinement criterion. If the criterion is not satisfied, the sampling rate is increased by generating more samples using the refinement strategy. This procedure is repeated until the refinement criterion is met or some maximum number of iterations is reached. The goal of the refinement criterion is to estimate the local bandwidth of the image to control the sampling rate. Various measures have been proposed, e.g., based on the maximum difference of intensity [137] or contrast [90] in a small neighborhood of samples, or statistical estimates of the SNR (signal-to-noise ratio) [26] or the variance of the samples [64]. When new samples are introduced in the refinement step, it is important to preserve the spectral properties of the initial set of samples. Algorithms for constructing hierarchical point sets with blue-noise characteristics suitable for adaptive sampling were developed in [83] and in [51].

2.4.3 Discussion

Since most signals of interest are not band-limited, uniform sampling at a higher frequency and postfiltering will alleviate but not completely avoid aliasing. Moreover, increasing the sampling frequency leads to higher memory and computational requirements of most algorithms. Nonuniform sampling with blue-noise patterns can transform structured aliasing into high frequency noise, but such patterns are used predominantly in high quality offline renderers. Due to their irregularity, they are not suitable for interactive, hardware accelerated rendering pipelines.
In contrast, prefiltering is performed by applying a low-pass filter to the signal before sampling, hence it is the more theoretically justified antialiasing method. Using an ideal low-pass filter, the filtered signal can be band-limited to the Nyquist frequency of the sampling grid and thus aliasing is avoided completely. In practical interactive rendering pipelines, prefiltering is implemented as a convolution in the spatial domain; hence prefilters with a small support are desirable for efficiency reasons. Unfortunately, the ideal low pass filter has infinite support in the spatial domain, therefore it cannot be applied in such systems. Since the widths of a filter in the spatial and frequency domains are inversely related, some aliasing will be inevitable during sampling. This trade-off is discussed in more detail in Section 2.5. In Chapter 5, we present an efficient, high quality prefiltering approach to point-based rendering.

2.5 DIGITAL FILTERS FOR COMPUTER GRAPHICS

As has become clear from the previous sections, digital filters play a crucial role in computer graphics. Although the ideal low-pass or ideal reconstruction filter has the ability to exactly band-limit a signal to the Nyquist frequency of a sampling grid, or to perfectly reconstruct a sampled signal, respectively, it is not useful in practice because of its infinite support in the spatial domain. For efficiency reasons, we strive to design digital filters with a small support in the spatial domain that approximate the spectral properties of the ideal filter. Low-pass filter characteristics are usually described in terms of a pass band, a transition band, and a stop band, as illustrated in Figure 2.7. Typically, the filter is constructed such that the Nyquist limit of the sampling grid is in the center of the transition band.

![Figure 2.7](image)

**Figure 2.7** Low-pass filter characterization in terms of a pass band $|\omega| < |\omega_p|$, transition band $|\omega| < |\omega_s|$, and stop band $|\omega| < |\omega|$. The filter is designed such that the Nyquist limit $\omega_N$ is in the center of the transition band.

Independent of their use as low-pass or reconstruction filters, we characterize the filters by a set of criteria important for computer graphics applications, including blurring, reconstruction error or post-aliasing, flat-field response, anisotropic effects, and ringing.

If the filter attenuates high frequencies in the pass band (or baseband), fine detail in the filtered image is lost, the image is blurry. In contrast, if the filter has a leak of high
frequencies above the Nyquist limit, aliasing occurs in the processed image, which is called post-aliasing.

Pavicic [104] proposed to measure the quality of a filter by its ability to reconstruct a flat (i.e., constant, all the samples have the same value) signal. He uses the error between the flat-field response of the filter and the constant signal as a filter quality criterion. Note that this error is zero for any weighted-average filter (see Section 2.7). The oscillating intensity patterns that occur if a filter does not have a perfect flat field response are sometimes also referred to as sample-frequency ripple.

Anisotropic effects occur when a filter has unequal response in different spatial directions. For example, separable filters suffer from this problem when used on regular sampling grids, the underlying square pixels showing through in the reconstructed image. It is clear that filters with a radially symmetric impulse response do not suffer from this problem.

Ringing appears when a sharp edge turns into a rippling set of lighter and darker bands. This effect results from alternating positive and negative lobes in the filter response. In particular, the ideal low-pass filter, which is a sinc function, suffers from this problem when applied to finite images.

A more detailed discussion of filter design for image synthesis can be found in [91]. For a good overview of various practical filters, please refer to [42]. In the next section, we focus on Gaussian filters, since these are key to our techniques described in Chapters 5 and 7.

2.6 GAUSSIAN FILTERS

Gaussian functions play an important role in many areas of applied mathematics, in particular in statistics. For digital signal processing they are attractive because they provide a unique combination of reasonable spectral characteristics and analytical properties. In Section 2.6.1, we introduce one-dimensional Gaussians and analyze their filter characteristics. It is straightforward to generalize Gaussians to higher dimensions, as is shown in Section 2.6.2. In Section 2.6.3, we summarize the analytical properties that are prerequisite for our techniques described in Section 2.7.4.

2.6.1 One-Dimensional Gaussian Filters

A one-dimensional Gaussian function is defined as

\[ g_{\sigma^2}(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}, \]

where \( \sigma^2 \) is called the variance, and \( \sigma \) the standard deviation. In this form, the Gaussian is normalized to have a unit integral, i.e.,

\[ \int_{-\infty}^{\infty} g_{\sigma^2}(x) dx = 1. \]  

(2.10)

The frequency domain representation of this function is given by
hence the Fourier transform of the Gaussian is another Gaussian.

Figure 2.8 illustrates the relation of 1D Gaussians in the spatial and the frequency domain. A narrow impulse response in the spatial domain corresponds to a wide response in the frequency domain, and vice versa. For comparison, the box function of an ideal low-pass filter in the frequency domain is shown, too. Assuming that the sampling grid has unit spacing, the cutoff frequency is at $\omega = 1/2$, hence band-limiting the signal to the Nyquist limit of the grid. Clearly, choosing a suitable standard deviation $\sigma$ is a trade-off between blurriness and aliasing in the output signal, i.e., a trade-off between a deficiency of high frequencies in the pass band and leakage in the stop band. For image processing applications, values between $1.0 < \sigma < 2.0$ are often appropriate.

Gaussian filters belong to the class of IIR (infinite impulse response) filters, since they do not have a compact support in the spatial domain. In practice, Gaussian filters are windowed, i.e., multiplied with a box, to truncate them to a compact support. Obviously, this changes the spectral properties of the filter. But this is unproblematic, since the Gaussian function decays quickly to zero. The box window typically has a width of $[-3\sigma^2, 3\sigma^2]$ to $[-5\sigma^2, 5\sigma^2]$, the value of the Gaussian at the boundary of these intervals being $4.4e-3$ and $1.5e-6$, respectively.

Figure 2.9 compares the flat field response of Gaussians with varying standard deviation. The flat field response $f_{\sigma}(x)$ is computed as the output signal reconstructed from samples with value one placed at unit intervals, i.e.,
2.6 Gaussian Filters

2.6.1 Gaussian Filters

We also compute the corresponding squared per sample reconstruction error as a function of the standard deviation $\sigma$:

$$
e(\sigma) = \int_{-0.5}^{0.5} (f_{\sigma}(x) - 1)^2 dx,
$$

(2.13)

a pixel being in the range $-0.5 < x < 0.5$. For standard deviations $\sigma = 1/3$ and $\sigma = 1/2$, the flat field response exhibits heavy oscillations at the frequency of the pixel grid (also known as sample frequency rippling), and the reconstruction errors $e(1/3) = 2.25e - 2$ and $e(1/2) = 1.0e3 - 4$ are significant. However, for $\sigma = 1$, which is useful for image processing as we argued above, the flat field response is almost constant. The corresponding reconstruction error $e(1) = 1.43e - 17$ is so small that visual artifacts do not appear in practice.

![Flat field response of 1D Gaussians with varying standard deviation.](FIGURE 2.9)

2.6.2 Multidimensional Gaussian Filters and Their Analytical Properties

It is straightforward to generalize Gaussians to higher dimensions. The $N$-dimensional Gaussian is defined as

$$
g_N^V(x) = \frac{1}{(2\pi)^{N/2}|V|^{1/2}}e^{-\frac{1}{2}x^TV^{-1}x}.
$$

(2.14)

Here, $V$ is the symmetric $N \times N$ variance matrix, with $|V|$ its determinant, and $x$ is a $N \times 1$ column vector. $V$ is called the variance matrix because it plays a role analogous to the scalar variance $\sigma^2$ of a 1D Gaussian. In this form, the $N$-dimensional Gaussian is normalized to unit integral, i.e.,
The Fourier transform $G_N^N(\omega)$ of an $N$-dimensional Gaussian $g_N^N(x)$ is again an $N$-dimensional Gaussian. However, note that the Gaussian in the frequency domain is not normalized to unit integral:

$$G_N^N(\omega) = e^{\frac{-1}{2}\omega^T V \omega} = (2\pi)^{N/2}|V|^{1/2}G_N^{N-1}(\omega).$$  \hspace{1cm} (2.16)

Gaussians offer a number of analytical properties that make them attractive as digital filters for image synthesis. We summarize how operations such as linear mapping, convolution, and integration, can be evaluated efficiently on Gaussians:

**Linear mapping.** An $N$-dimensional linear mapping is defined as $y = Mx$, where $M$ is an $N \times N$ matrix, and $y$ and $x$ are $N \times 1$ column vectors. We apply this mapping to the Gaussian $g_N^N(x)$ by substituting $x = M^{-1}y$, yielding:

$$g_N^N(M^{-1}y) = \frac{1}{(2\pi)^{N/2}|V|^{1/2}} \cdot e^{\frac{1}{2}(M^{-1}y)^T V^{-1}(M^{-1}y)} = \frac{|M|}{(2\pi)^{N/2}|MVM^T|^{1/2}} \cdot e^{\frac{1}{2}y^T (M^{-1}V^{-1}M^{-1})y}.$$  \hspace{1cm} (2.17)

Hence, under the linear mapping $M$, the Gaussian with variance matrix $V$ is transformed into a Gaussian with variance matrix $MVM^T$. However, the transformed Gaussian is not normalized to unit integral anymore, but it is scaled with the determinant $|M|$ (see also [49]).

**Convolution.** The convolution of two Gaussians is easily computed in the frequency domain, since convolution in the spatial domain corresponds to multiplication in the frequency domain [49]. With Equation 2.16, we have

$$g_N^N(x) \otimes g_N^N(x) \leftrightarrow G_N^N(\omega) \cdot G_N^N(\omega) = e^{\frac{1}{2}\omega^T V \omega} \cdot e^{\frac{1}{2}\omega^T W \omega} = e^{\frac{1}{2}\omega^T (V+W) \omega} = G_{V+W}(\omega).$$  \hspace{1cm} (2.18)

Therefore, variances are added when two Gaussians are convolved:

$$g_N^N(x) \otimes g_N^N(x) = g_N^N(x + w).$$  \hspace{1cm} (2.19)

**Integration.** Multidimensional Gaussian functions play an important role in mathematical statistics, where they are known as multivariate normal distributions. An important result from this field is that all subsets of a multivariate, normally distributed random variable are themselves normally distributed. This result is equivalent to the
fact that the integration of an $N$-dimensional Gaussian along one coordinate axis results in an $(N-1)$-dimensional Gaussian, i.e.,

$$\int_{-\infty}^{\infty} g_N^V(x)dx_i = g_{N-1}^V(x),$$  \hspace{1cm} (2.20)

where $0 \leq i < N$ and the vector $x$ has dimensions $(N-1) \times 1$. The new variance matrix $\bar{V}$, which is $(N-1) \times (N-1)$, is obtained by skipping the $i$-th row and column in the original variance matrix $V$.

### 2.6.3 Two-Dimensional Gaussians for Image Processing

For image processing applications, we are most interested in the 2D Gaussian, i.e.,

$$g_2^V(x) = \frac{1}{2\pi|V|^{1/2}}e^{-\frac{1}{2}x^TV^{-1}x},$$ \hspace{1cm} (2.21)

where $V$ is a symmetric $2 \times 2$ matrix and $x$ is a $2 \times 1$ column vector.

The isocontours of the 2D Gaussian $g_2^V$ are described by the quadratic form

$$\frac{1}{2}x^TV^{-1}x = r^2,$$ \hspace{1cm} (2.22)

where $r \in \mathbb{R}$ is a constant. Equation 2.22 is an implicit quadratic polynomial, representing a conic section. Without matrix notation, Equation 2.22 expands to

$$\frac{1}{2}(A'x_0^2 + B'x_0x_1 + C'x_1^2) = r^2,$$ \hspace{1cm} (2.23)

where $x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$ and $V^{-1} = \begin{bmatrix} A' & B'/2 \\ B'/2 & C' \end{bmatrix}$.

Depending on whether $|V|$ is positive, zero, or negative, the conic section has the shape of an ellipse, a parabola, or a hyperbola respectively. Hence we also call the 2D Gaussian $g_2^V$ an elliptical Gaussian. For more details on conic sections, please refer to [16].

In 2D, the eigenvectors of $V$ determine the principal axes of the ellipse represented by Equation 2.22, and the eigenvalues determine the variances along these axes. In particular, if $V$ is a uniform scaling matrix, i.e., $V = \text{diag}(\sigma^2, \sigma^2)$, the implicit quadratic polynomial 2.22 can be written as

$$\frac{1}{2}\left(\frac{1}{\sigma^2}x_0^2 + \frac{1}{\sigma^2}x_1^2\right) = r^2.$$ \hspace{1cm} (2.24)

The isocontours of 2.24 are circles and the corresponding 2D Gaussian is radially symmetric. Each cross section is a 1D Gaussian with variance $\sigma^2$. Therefore, a practical choice for image processing on regular 2D grids is $V = \text{diag}(\sigma^2, \sigma^2)$ where $1.0 < \sigma < 2.0$ (see Section 2.6.1).

The Fourier transformation of a 2D elliptical Gaussian is, analogous to the 1D case, an elliptical 2D Gaussian again:
Hence, we refer to Section 2.6.1 for a discussion of the spectral properties of Gaussian functions. As an example, Figure 2.10 shows an anisotropic 2D elliptical Gaussian with variance matrix

\[
V = \begin{bmatrix}
0.504 & 0.318 \\
0.318 & 0.796 
\end{bmatrix}
\]

in the spatial (Figure 2.10a) and the frequency domain (Figure 2.10b). Note that along each direction, a narrow impulse response in the spatial domain corresponds to a wide response in the frequency domain.

As in the 1D case, the 2D elliptical Gaussian is an IIR filter. Truncation to a finite support is therefore necessary for practical use. Usually, truncation is performed as follows by setting

\[
g^2_v(x) = 0 \text{ for } \{x \mid x^T V^{-1} x > r^2\}.
\]

Therefore, the shape of the truncated Gaussian filter is an ellipse. For \( r \), values between 2.0 < \( r \) < 4.0 lead to good results. Figure 2.11 shows a comparison of an image reconstruction operation with a radially symmetric 2D Gaussian (\( \sigma = 1 \)) and cutoff radii \( r = 1 \) and \( r = 3 \). Clearly, reconstruction errors are visible in Figure 2.11a, while the image in Figure 2.11b is reconstructed smoothly; no artifacts due to truncation appear.

### 2.7 RESAMPLING FILTERS

In Section 2.4, we described how a continuous 2D image function \( g_c(x) \) can be filtered to generate a discrete image without aliasing artifacts using either a pre- or post-filtering approach. However, we left open how to compute the continuous function
2.7 RESAMPLING FILTERS

On the one hand, we pointed out that in general it is impossible to get an explicit representation of $g_c(x)$. In this case, a postfiltering method has to be employed (see Section 2.4.2). On the other hand, for certain restricted operations, such as image warping and texture mapping, the output image function $g_c(x)$ can be derived explicitly. In this section, we show how to combine an explicit representation of $g_c(x)$ with a band-limiting step, resulting in a prefiltering pipeline.

As described by Heckbert [49], the task of image warping and texture mapping essentially consists of mapping a 2D signal (i.e., an image) from a source to a destination domain. The mapping is arbitrary, and it usually involves geometric distortions of the image. Well known mappings are affine, bilinear, and projective mappings. Since we work with discrete images in practice, image warping involves mapping a discrete function defined at positions $u_k$ in the source domain to the destination domain, and sampling the warped function at discrete positions $x_k$ in the destination domain. It is important to realize that the positions $u_k$ and $x_k$ are not identical in general, hence the image has to be resampled in this process. In this section, we will present the concept of resampling filters for this purpose. Resampling filters combine the reconstruction of a continuous input signal, the mapping from source to destination domain, and prefiltering the continuous output function $g_c(x)$ before sampling at the output positions $x_k$ to avoid aliasing artifacts.

2.7.1 Destination Domain Resampling Filters

As illustrated in Figure 2.12, resampling consists of four steps: reconstruction, warping, prefiltering, and sampling [49].
1. In the first step, we reconstruct a continuous signal from the discrete input signal. Given the sampled function values \( f(u_k) \) and reconstruction filters \( r_k(u) \) in source space, the continuous input function \( f_c(u) \) is

\[
f_c(u) = \sum_k f(u_k) r_k(u - u_k),
\]

(2.28)

where \( u \) are continuous source domain coordinates. Equation 2.28 corresponds to the adaptive local filtering approach, which is described in more detail in Section 3.1.2. Here, the \( u_k \) are *nonuniform* sampling positions, and the reconstruction kernels \( r_k \) may be different for each sample.

2. We denote the mapping from source to destination domain by \( x = m(u) \), and \( u = m^{-1}(x) \) is its inverse. Here, \( x \) denotes continuous destination space coordinates. Applying this mapping to the input function yields the continuous function \( g_c(x) \) in the destination domain:

\[
g_c(x) = (f_c \cdot m^{-1})(x) = f_c(m^{-1}(x)).
\]

(2.29)
3. Now the warped, continuous signal is prefiltered using a low-pass filter \( h \), resulting in the continuous output function \( g_c'(x) \):

\[
g_c'(x) = g_c(x) \ast h(x) = \int_{\mathbb{R}^2} g_c(\xi)h(x - \xi)d\xi.
\]

(2.30)

4. Finally, the continuous output signal is sampled by multiplying it with an impulse train \( i \) to produce the discrete output \( g(x) \):

\[
g(x) = g_c'(x)i(x).
\]

(2.31)

This procedure suggests a multi-pass approach, in which we first reconstruct, then warp, and convolve the input signal. However, we can avoid the explicit construction of the continuous signal by reordering the above operations. We derive an expression for the warped continuous output function by expanding the operations in reverse order:

\[
g_c'(x) = \int_{\mathbb{R}^2} h(x - \xi) \sum_k f(u_k)r_k(m^{-1}(\xi) - u_k)d\xi
\]

\[
= \sum_k f(u_k)\rho_k(x),
\]

(2.32)

where

\[
\rho_k(x) = \int_{\mathbb{R}^2} h(x - \xi)r_k(m^{-1}(\xi) - u_k)d\xi.
\]

(2.33)

We call the warped and filtered reconstruction kernel \( \rho_k(x) \) a resampling filter, which is expressed as a destination space integral here. Equation 2.32 states that we can first warp and filter each reconstruction kernel \( r_k \) individually to construct the resampling filters \( \rho_k \) and then sum up the contributions of these filters in destination space.

Unfortunately, it is difficult to evaluate the integral in Equation 2.33 in the presence of a general mapping function \( m^{-1} \). To simplify this integral, we replace the general mapping \( m(u) \) by its local affine approximation \( m_k(u) \) at a point \( u \):

\[
m_k(u) = x_k + J_k \cdot (u - u_k),
\]

(2.34)

where \( x_k = m(u_k) \) and the Jacobian \( J_k = \frac{\partial}{\partial u} m(u_k) \).

(2.35)

In the case of 2D image warping, the Jacobian \( J_k \) is a \( 2 \times 2 \) matrix of partial derivatives of \( m(u) \):

\[
\frac{\partial m}{\partial u} = \begin{bmatrix}
\frac{\partial x_0}{\partial u_0} & \frac{\partial x_0}{\partial u_1} \\
\frac{\partial x_1}{\partial u_0} & \frac{\partial x_1}{\partial u_1}
\end{bmatrix}, \text{ where } u = \begin{bmatrix} u_0 \\ u_1 \end{bmatrix} \text{ and } x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}
\]

(2.36)

are column vectors.
The approximation 2.35 is appropriate, because in practice we work with reconstruction filters that have a compact support. We use the approximation only in a small neighborhood around $\mathbf{u}_k$, where it is most accurate. Therefore we have not seen any visual artifacts due to this approximation.

We use Equation 2.35 to rearrange Equation 2.33, exploiting that the affine mapping of Equation 2.34 has the property that $m_k^{-1} (\mathbf{u}) - m_k^{-1} (\mathbf{v}) = J_k^{-1} \cdot (\mathbf{u} - \mathbf{v})$:

$$
\rho_k(\mathbf{x}) \approx \int_{\mathbb{R}^2} h(\mathbf{x} - \xi) r'_k(m_k^{-1}(\xi) - \mathbf{u}_k) d\xi
$$

$$
= \int_{\mathbb{R}^2} h(\mathbf{x} - \xi) r'_k(m_k^{-1}(\xi) - m_k^{-1}(m_k(\mathbf{u}_k))) d\xi
$$

$$
= \int_{\mathbb{R}^2} h(\mathbf{x} - \xi) r'_k(J_k^{-1} \cdot (\xi - m_k(\mathbf{u}_k))) d\xi
$$

$$
= \int_{\mathbb{R}^2} h(\mathbf{x} - m_k(\mathbf{u}_k) - \tau) r'_k(\tau) d\tau
$$

$$
= (r'_k \otimes h)(\mathbf{x} - m_k(\mathbf{u}_k)),
$$

(2.37)

where

$$
r'_k(\mathbf{x}) = r_k(J_k^{-1} \mathbf{x})
$$

(2.38)

is a reconstruction kernel warped to destination space, and where we substituted $\tau = \xi - m_k(\mathbf{u}_k)$. According to the last equality in Equation 2.37, the resampling filter $\rho_k(\mathbf{x})$ is a convolution of a warped reconstruction filter and the low-pass filter. This is essential for the derivation of Gaussian resampling filters as presented in Section 2.7.4.

### 2.7.2 Source Domain Resampling Filters

Above, we derived a resampling filter that is defined in the destination domain. However, it is also possible to represent the resampling filter in source space as described by Heckbert [49]. We find the source space formulation of the resampling filter by mapping the prefilter $h$ to the source domain instead of mapping the reconstruction filter $r_k$ to the destination domain. This idea can be expressed mathematically by substituting $\xi = m(\mathbf{u})$ in Equation 2.33, yielding:

$$
\rho_k(\mathbf{x}) = \int_{\mathbb{R}^2} h(\mathbf{x} - m(\mathbf{u})) r'_k(\mathbf{u} - \mathbf{u}_k) \left| \frac{\partial m}{\partial \mathbf{u}} \right| d\mathbf{u},
$$

(2.39)

where $|\partial m / \partial \mathbf{u}|$ is the determinant of the Jacobian (Equation 2.36) and $d\xi = |\partial m / \partial \mathbf{u}| d\mathbf{u}$.

Similar to Equation 2.37, we use the affine approximation of the mapping to simplify the integral in Equation 2.39:
\[
\rho_k(x) = \int h(x - m_k(u))r_k(u - u_k)|J_k| du \\
= \int h(m_k^{-1}(x)) - m_k(u))r_k(u - u_k)|J_k| du \\
= \int h(J_k \cdot m_k^{-1}(x) - u)|r_k(u - u_k)|J_k| du \\
= \int h_k'(m_k^{-1}(x) - u)r_k(u - u_k) du \\
= (h_k' \otimes r)(m_k^{-1}(x) - u_k),
\]

where

\[
h_k'(u) = h(J_k u)|J_k|
\]

is the prefilter that is warped to the source domain.

### 2.7.3 Resampling Filter Properties

Summarizing the results from Sections 2.7.1 and 2.7.2, we have found that resampling, which is the process of warping, filtering, and sampling a discrete input signal, can be expressed as a weighted sum of sample values \( f(u_k) \) and resampling filters \( \rho_k \):

\[
g'_c(x) = \sum_k f(u_k) \rho_k(x). \quad (2.42)
\]

We have derived two alternative representations of the resampling filter, a destination space and a source space formulation, which are **mathematically equivalent**:

\[
\rho_k(x) = \int h(x - \xi)r_k(m_k^{-1}(\xi) - u_k) d\xi \\
= \int h(x - m(u))r_k(u - u_k) \left| \frac{\partial m}{\partial u} \right| du, \quad (2.43)
\]

Since the resampling filters are adaptive to the mapping being performed, they are different for each sample position. Therefore, the resulting filtering scheme is **space variant** and the weighted sum of Equation 2.42 does not correspond to a convolution. It depends on the application, whether the source or destination space formulation is more suitable. The source space formulation has been developed for texture mapping [49], whereas the destination space formulation is appropriate for splatting approaches [149].

**Ideal Resampling Filters.** If we use sinc functions as low-pass and reconstruction filters, Equation 2.43 represents an **ideal resampling filter**. Ideal resampling filters have an infinite support and arbitrary shape in general and process the input signal without introducing blurriness or aliasing. In practice, resampling filters are approximations of the ideal, with finite support and a restricted class of shapes, to enable efficient evalu-
ation. In the following, we discuss some common approximations of the low-pass filter, the reconstruction filter, and the mapping, which are often crude to make the resampling filter practical.

**Magnification and Minification Filters.** The integral of Equation 2.43 can be simplified drastically by assuming that either the reconstruction filter or the low-pass filter is an impulse.

When the mapping magnifies the input signal, the size and shape of the resampling filter is dominated by the reconstruction filter. In this case it is reasonable to assume that the low-pass filter is an impulse \( h = \delta \). Hence, we get a pure magnification filter

\[
\rho_k(x) = r_k(m^{-1}(x) - u_k),
\]

which only consists of a reconstruction filter that is warped to the destination domain.

In contrast, when the mapping minifies the input signal, the size and shape of the resampling filter is dominated by the low-pass filter. In this case, we could neglect the reconstruction filter and use an impulse instead, yielding a resampling filter

\[
\rho_k(x) = \left[ \frac{\partial m}{\partial u} \right] h(x - m(u_k)),
\]

which is a pure low-pass filter centered at the sample position \( m(u_k) \) in destination space. We call this a minification or decimation filter. Note that the shape of the filter is independent of the mapping and that it is scaled proportionally to the scale factor of the mapping.

A common approach for resampling is to switch between the magnification and the minification filter based on the scale factor of the mapping. However, in 2D it is not possible in general to determine unambiguously whether the mapping magnifies or minifies the input signal. Instead, the mapping can anisotropically magnify in one direction and minify in another at the same time. To solve this problem, we describe a class of unifying resampling filters in Section 2.7.4 that provides a smooth transition between magnification and minification, even in the presence of anisotropic scaling.

**Local Affine Approximation.** By introducing local affine approximations \( m_k \) to the mapping \( m \), we expressed both variants of the resampling filter as a convolution:

\[
\rho_k(x) \approx (r_k' \otimes h)(x - m_k(u_k)),
\]

\[
\rho_k(x) \approx (h_k' \otimes r)(m_k^{-1}(x) - u_k).
\]

While Equation 2.46 is a convolution of a warped reconstruction filter and a low-pass filter in destination space, Equation 2.47 is a convolution of a warped low-pass filter and a reconstruction filter in source space. Since the local affine approximation of the mapping is evaluated at each sample position \( u_k \), this scheme is still space variant in general. Only if the mapping \( m \) is globally affine, its Jacobian is constant and the resampling filter is space invariant.

### 2.7.4 Gaussian Resampling Filters

In this section, we will exploit the analytical properties of Gaussian filters (see Section 2.6) to derive an explicit expression of the resampling filter as represented by
Equations 2.46 and 2.47. We will use 2D Gaussians, although the procedure can be applied to Gaussians of any dimension.

The Gaussian resampling filter is based on the choice of Gaussians as low-pass filter and as reconstruction filters. Furthermore, our derivation relies on the local affine approximation of the mapping function, i.e., on the approximations of Equations 2.46 and 2.47 of the ideal resampling filter.

Let us denote the Gaussian reconstruction filters in source space by

$$r_k(u) = g_{R_k}(u) = \frac{1}{2\pi|R_k|^{1/2}} e^{-\frac{1}{2}u^T R_k^{-1} u},$$  \hspace{1cm} (2.48)

where $R_k$ is a $2 \times 2$ variance matrix. In general, $R_k$ is chosen according to the local sample distribution around each sample $u_k$, as described in Section 3.1.2. Similarly, the Gaussian low-pass filter in the destination domain is defined as

$$h(x) = g_{H}(x) = \frac{1}{2\pi|H|^{1/2}} e^{-\frac{1}{2}x^T H^{-1} x},$$  \hspace{1cm} (2.49)

with its $2 \times 2$ variance matrix $H$, which is determined by the sampling grid in destination space. Usually, this is a regular grid with unit spacing, therefore an appropriate choice for $H$ is the identity matrix (see also Section 2.6).

**The Gaussian Destination Space Resampling Filter.** To construct the Gaussian destination space resampling filter, we need to map the Gaussian reconstruction kernels to the destination domain as in Equation 2.38:

$$r'_k(x) = r_k(J_k^{-1} x) = \frac{1}{|J_k|} g_{J_k R_k J_k^T}(x),$$  \hspace{1cm} (2.50)

where we used Equation 2.17 to compute the linear warp of the Gaussian. Substituting Equations 2.49 and 2.50 into Equation 2.46 yields the Gaussian resampling filter:

$$\rho_k(x) \approx (r'_k \otimes h)(x - m_k(u_k))$$

$$= \frac{1}{|J_k|} \left( g_{J_k R_k J_k^T} \otimes g_{H} \right)(x - m_k(u_k))$$

$$= \frac{1}{|J_k|} g_{J_k R_k J_k^T + H}(x - m_k(u_k)).$$  \hspace{1cm} (2.51)

Here, we applied Equation 2.19 to compute the convolution of the two Gaussians. This equation represents the Gaussian destination space resampling filter. Hence, if we use Gaussians as reconstruction and low-pass filters, the resampling filter is a Gaussian too. Therefore, it is also called an elliptical weighted average (EWA) resampling filter. Its variance matrix is easily computed as $J_k R_k J_k^T + H$, which requires only two matrix multiplications and one addition.

**The Gaussian Source Space Resampling Filter.** To compute the Gaussian resampling filter in source space, we start from Equation 2.47. Hence we need to map the
low-pass filter to source space (Equation 2.41), which is accomplished using Equation 2.17 as above:

\[
h'_k(\mathbf{u}) = |\mathbf{J}_k| h(\mathbf{J}_k \mathbf{u}) = g \mathbf{J}_k^T \mathbf{H} \mathbf{J}_k^T (\mathbf{u}). \tag{2.52}
\]

We now substitute Equations 2.48 and 2.52 into Equation 2.47:

\[
\rho_k(x) \approx (h'_k \otimes r)(m_k^{-1}(x) - \mathbf{u}_k) = \left(g \mathbf{J}_k^T \mathbf{H} \mathbf{J}_k^T \otimes g \mathbf{R}_k\right)(m_k^{-1}(x) - \mathbf{u}_k)
= g(\mathbf{R}_k + \mathbf{J}_k^T \mathbf{H} \mathbf{J}_k^T)(m_k^{-1}(x) - \mathbf{u}_k),
\tag{2.53}
\]

yielding the Gaussian source space resampling filter. Also in this formulation, the resampling filter is a single Gaussian, with variance \( \mathbf{R}_k + \mathbf{J}_k^{-1} \mathbf{H} \mathbf{J}_k^{-1\top} \).

**Gaussian Resampling Filter Properties.** It is easy to verify that the source and destination space formulations of the Gaussian resampling filter are mathematically equivalent. To show this, we use the identity

\[
\mathbf{x} - m_k(\mathbf{u}_k) = m_k(m_k^{-1}(\mathbf{x})) - m_k(\mathbf{u}_k) = \mathbf{J}_k \cdot (m_k^{-1}(\mathbf{x}) - \mathbf{u}_k), \tag{2.54}
\]

and substitute this into the destination space formulation:

\[
\frac{1}{|\mathbf{J}_k^{-1}|} g \mathbf{J}_k \mathbf{R}_k \mathbf{J}_k^T + \mathbf{H}(\mathbf{x} - m_k(\mathbf{u}_k)) = \frac{1}{|\mathbf{J}_k^{-1}|} g \mathbf{J}_k \mathbf{R}_k \mathbf{J}_k^T + \mathbf{H}(\mathbf{J}_k \cdot (m_k^{-1}(\mathbf{x}) - \mathbf{u}_k))
= g(\mathbf{R}_k + \mathbf{J}_k^T \mathbf{H} \mathbf{J}_k^T)(m_k^{-1}(\mathbf{x}) - \mathbf{u}_k),
\tag{2.55}
\]

yielding the source space formulation. This procedure is illustrated in Figure 2.13. The elliptical resampling filter is visualized using isocontours in source space to the left and in destination space to the right. As an example, the resampling of a checkerboard texture is shown in the lower images.

In Figure 2.14, we illustrate the behavior of Gaussian resampling filters in destination space in different situations. Note that in destination space, the low-pass filter always has the same shape, since it only depends on the output sampling grid, but it is independent of the mapping function. Under minification, as shown to the left, the filter shape is dominated by the low-pass filter. Considering the variance matrix \( \mathbf{J}_k \mathbf{R}_k \mathbf{J}_k^T + \mathbf{H} \) of the resampling filter in destination space (Equation 2.51), this is intuitively clear: when the mapping is minifying the input function, its Jacobian scales down the reconstruction filter, and the variance matrix is dominated by the contribution of the low-pass filter. On the other hand, the resampling filter is largely determined by the reconstruction filter under magnification, as shown in the middle of Figure 2.14. In this case, the Jacobian scales up the reconstruction filter, and the variance matrix of the resampling filter is dominated by the enlarged reconstruction filter. Moreover under anisotropic minification-magnification, the Jacobian anisotropically scales the reconstruction filter, as shown to the left of Figure 2.14. The scaling guarantees that
the resampling filter is always wider than both the reconstruction and low-pass filter, so it is impossible that the filter falls between samples of the output grid.

In summary, Gaussian resampling filters provide a unified scheme for reconstruction and low-pass filtering. Since Gaussians are closed under affine mappings and convolution, Gaussian resampling filters can be constructed elegantly by combining Gaussian reconstruction and Gaussian low-pass filtering. They may be used for non-affine mappings by using the local affine approximation to the mapping as determined by its Jacobian. Gaussian resampling is ideal to the extent that the Gaussian is an ideal low-pass filter and the mapping is affine. Equivalent formulations for the filter in source and in destination space can be derived.

**FIGURE 2.13** Source (left) and destination space (right) formulation of the Gaussian resampling filter visualized using isocontours. As an example, the lower images show the resampling of a checkerboard texture.
**FIGURE 2.14** The behavior of Gaussian resampling filters in destination space under minification (left), magnification (middle), and anisotropic minification-magnification (right).
The theory of nonuniform sampling and reconstruction is by far not as complete as the theory of uniformly distributed samples. Also, the nonuniform reconstruction problem is often viewed in a more general perspective than from the purely signal processing based point of view. In this broader context, it is also called the scattered data approximation problem. For uniform sampling, signal processing theory provides a precise criterion when a continuous signal can be sampled and reconstructed without introducing any error. Moreover, efficient procedures for sampling and reconstruction can be derived using Fourier analysis. Unfortunately, these results cannot be generalized easily to nonuniform sampling.

Lacking a unified theory for nonuniform sampling and reconstruction, a variety of methods and algorithms has been developed, designed for different applications with different requirements. The approaches used in computer graphics include radial basis function techniques [115, 20], iterative methods [36, 144], variational techniques [129], and multistep or hierarchical approaches [18, 43, 90]. Note that there exists a vast body of literature for each of those fields, hence the above references are intended to serve merely as entry points. We refer to [42] for a pragmatic overview of methods developed specifically for computer graphics applications.

Many of the sophisticated approaches mentioned above are computationally expensive and cannot handle hundreds of thousands of samples in an interactive system. In this section, we review the local filtering approach (Section 3.1) to nonuniform reconstruction. This technique is well suited for interactive applications due to its effectiveness and simplicity. We use local filtering as a basis for our surface representation (Chapter 4) and for our point-based rendering algorithm (Chapter 5).
3.1 LOCAL FILTERING

Glassner [42] uses the term local filtering to denote continuous reconstruction of non-uniformly sampled signals by directly applying a reconstruction filter, as it is done with uniform samples. If we use the same reconstruction filter for all samples, this is called space invariant local filtering (Section 3.1.1) and it amounts to a convolution of the sampled signal with the reconstruction filter. In adaptive local filtering (Section 3.1.2), we use different reconstruction kernels for each sample, which are adapted to the local sampling distribution. Furthermore in Section 3.1.3, we discuss sampling strategies for continuous signals under the assumption that reconstruction will be performed by local filtering with a given reconstruction filter.

3.1.1 Space Invariant Local Filtering

In this section we consider the problem of reconstructing a continuous signal $g$ from a given set of samples $f(x_i)$ of an unknown signal $f$ at nonuniform sampling locations $x_i$. For example, this problem arises in postfiltering image synthesis pipelines with nonuniform sampling (see Section 2.4.2), where the $f(x_i)$ are image samples that are computed by a rendering algorithm, e.g., by ray tracing [90]. Another typical scenario is that the $f(x_i)$ are acquired using laser scanners [70] or digital cameras [80].

Analogous to reconstruction from uniform samples, local filtering means that the continuous signal $g(x)$ is reconstructed from a nonuniformly sampled signal by convolving the sampled signal with a reconstruction kernel $h(x)$. Technically, the sampled signal $f(x)$ can be represented as a sum of scaled impulses:

$$\tilde{f}(x) = \sum_i f(x_i) \delta(x - x_i). \quad (3.1)$$

Therefore, the convolution of $\tilde{f}(x)$ with $h(x)$ reduces to a weighted sum of reconstruction kernels:

$$g(x) = \tilde{f}(x) \otimes h(x) = \sum_i f(x_i) \int_{\mathbb{R}} \delta(t - x_i) h(x - t) dt$$

$$= \sum_i f(x_i) h(x - x_i). \quad (3.2)$$

Since each sample $f(x_i)$ weights the same kernel $h$ in Equation 3.2, we call this process space-invariant filtering.

Of course, the reconstruction filter should have suitable spectral characteristics as described in Section 2.5. If the original signal $f$ was band-limited, the cutoff frequency of $h$ is chosen according to the maximum frequency in $f$ as with uniformly sampled signals. Otherwise, the reconstruction kernels should be determined adaptively to the sampling positions $x_i$, as described in Section 3.1.2.

However, simply ignoring the nonuniformity of the sampling pattern leads to undesired artifacts. For example, if we reconstruct a signal with constant value $f(x) = c$ that is sampled nonuniformly, i.e.,
we will always get intensity variations in the reconstruction \( g(x) \). If samples clump together in a small region, this area becomes too bright; regions where the samples are sparse will be too dark [90]. This is because the sum of the reconstruction kernels depends on the nonuniform sampling positions \( x_i \) and therefore is not constant. Clearly, it is not possible to achieve a perfect flat field response (see Section 2.5) with this local filtering scheme.

**Weighted Average Filtering.** A straightforward solution to this problem is to reconstruct the signal as a *weighted average* instead of a weighted sum of reconstruction kernels. In other words, we *normalize* the reconstructed value at each position \( x \) by dividing by the sum of the reconstruction kernels \( \sum_i h(x - x_i) \) at the point \( x \), i.e.,

\[
\sum_i f(x_i)h(x - x_i) \\
g(x) = \frac{\sum_i h(x - x_i)}{\sum_i h(x - x_i)}.
\]  

This scheme is also called *weighted average filtering* [42]; it has been used in a number of implementations, e.g., [24, 26]. Clearly, because of the normalization division, it has a perfect flat field response, independent of the reconstruction kernel \( h \).

### 3.1.2 Adaptive Local Filtering

If the spectral properties of a nonuniformly sampled signal are unknown, e.g., the maximum frequency contained in the signal is not known, it is not clear from signal processing theory which are the appropriate parameters for a reconstruction filter. If the signal was uniformly sampled, a reasonable procedure would be to assume that the signal is sampled *properly*, i.e., that the sampling grid respects the Nyquist limit of the signal. Hence, we would conclude that the signal is bandlimited to half the frequency of the sampling grid and choose this as the cutoff frequency for the reconstruction filter. Unfortunately, if the sampling is nonuniform, there is no such simple relation between the sampling grid and the Nyquist frequency.

One idea to solve the problem is to analyze the distribution of sample points in a *small neighborhood* around each sample and to estimate the local sampling density in this neighborhood. Then, an appropriate reconstruction filter \( h_i \) is determined at each sample point \( x_i \) using this analysis. Reconstruction by local filtering is now performed by weighting each sample \( f(x_i) \) with its corresponding local kernel \( h_i \), i.e., Equation 3.2 becomes

\[
g(x) = \sum_i f(x_i)h_i(x - x_i),
\]

which we call *adaptive local filtering*. In contrast to Equation 3.2, this scheme is not a convolution, since the reconstruction kernels \( h_i \) are different for each sample point; the filtering scheme is *space variant*. 

\[
g(x) = \sum_i c \cdot h(x - x_i) = c \cdot \sum_i h(x - x_i),
\]
We can also generalize adaptive local filtering (Equation 3.5) to include normaliza-
\[ g(x) = \frac{\sum f(x_i)h_i(x - x_i)}{\sum h_i(x - x_i)}, \tag{3.6} \]
which we call *adaptive weighted average filtering*.

Given a neighborhood of nonuniform sampling positions, but no information about
the sampled signal, there is no precise theory that explains how to choose the correct
local reconstruction kernels \( h_i \). In the following we discuss two heuristics for deter-
mining appropriate kernels, focusing on Gaussian kernels that are truncated to have
local support.

**Adaptive Gaussian Filters.** Given a nonuniform 2D sampling grid, we want to com-
pute elliptical Gaussian filters, truncated to have local support and centered at each
sample position, that are appropriate for continuously reconstructing a signal sampled
on that grid using Equation 3.5 or Equation 3.6. Our goal in this process is to satisfy
the following two conditions:

- First, the support of the kernels should be chosen such that any point in the convex
  hull of the sampling grid lies in the support area of at least one reconstruction ker-
  nel. This guarantees that the reconstructed signal does not contain any holes, i.e.,
  that it is defined at any point.

- Second, the shape and orientation of the Gaussians (i.e., their variance matrices)
  should be chosen such that the reconstruction does not suffer from aliasing prob-
  lems while avoiding excessive blurriness. Our approach to compute the variance
  matrices for nonuniform grids is inspired by the analysis of Gaussian filters for
  uniform reconstruction (see Section 2.6.1).

Our heuristics proceed as follows: In a first step, we compute ellipses centered at each
sample position such that they cover the sampling grid completely. In the second step,
we derive suitable variance matrices for the Gaussian kernels from the ellipses.

A first approach to cover the sampling grid with ellipses is based on constructing
the Voronoi diagram of the sampling positions \( x_i \), as illustrated in Figure 3.1. We com-
pure the minimum area rectangular bounding box at each position \( x_i \) that contains its
corresponding Voronoi cell. Then the ellipse at \( x_i \) is defined by choosing the edges of
the bounding box as its major and minor axis. For uniform grids, this procedure results
in circles whose radii correspond to the spacing of the grid (Figure 3.1a). Note that for
nonuniform grids (Figure 3.1b), the center of the bounding box does not necessarily
coincide with \( x_i \). Hence, we adjust the length of the ellipse radii by this displacement
to guarantee that the bounding box is covered completely.

Next, we need to determine suitable variance matrices for the elliptical Gaussian fil-
ters. For uniform grids with unit sample spacing, we found that Gaussians with a unit
variance, i.e., their variance matrix is the identity matrix \( I \), are a good compromise
between aliasing and blurriness. In this case, the four nearest neighbors of each grid-
point \( x_i \), which are at unit distance, lie on an isocontour of the exponent of the Gauss-
ian with isovalue \(-1/2\); hence the isocontour is
Therefore, we represent the ellipses computed above in their implicit form as isocurves of the exponent of the Gaussian filter with isovalue $-1/2$, i.e., the ellipses computed above are given by

$$\frac{1}{2}(x - x_i)^T I(x - x_i) = \frac{1}{2}. \quad (3.7)$$

where the $V_i$ are the variance matrices of the Gaussians. Given the major and minor axis $a = (a_0, a_1)^T$ and $b = (b_0, b_1)^T$ of an ellipse, it is easy to verify that, the corresponding variance matrix $V_i$ is

$$V_i = \begin{bmatrix} a & b \\ b & a \end{bmatrix}, \quad (3.9)$$

Hence the local Gaussian filters are defined as

$$h_i(x) = g_{V_i}(x) = \frac{1}{2\pi |V_i|^{1/2}} e^{-\frac{1}{2}V_i^{-1}x}.$$

We truncate these Gaussians $g_{V_i}$ to a finite support as discussed in Section 2.6.3 (Equation 2.27). In Figure 3.2 we illustrate our method on a jittered grid whose resolution in the horizontal direction is twice the resolution in the vertical direction. In Figure 3.2a we used a cutoff radius of $r = 0.25$ to emphasize the shape and orientation of the ellipses, in Figure 3.2b we set $r = 0.5$ to show the overlap of the filters.

Usually, we want to avoid the computation of the global Voronoi diagram of the sampling positions, because it is computationally too expensive (see also Chapter 4).
An alternative is to construct each Voronoi cell locally using the $k$-nearest neighbors around each point $x_i$. However, this provides only an approximation to the correct cell. If the sampling grid exhibits abrupt changes in the sampling density, $k$ has to be chosen larger to avoid holes in the reconstruction. The larger $k$, the more robust is this method, but also the more expensive to compute. For most grids, values of $20 \leq k \leq 30$ are sufficient.

An even more efficient alternative is to directly fit a rectangular bounding box to cover the $k$-nearest neighbors around each position $x_i$, without computing the Voronoi cell. However, the quality of the result depends on an appropriate choice of $k$. On the one hand, values in the range $8 \leq k \leq 12$ lead to good results for rather uniform sampling distributions. On the other hand, a constant number $k$ is not appropriate for distributions with abrupt changes in the sampling density. An alternative is to choose $k$ adaptively. For example Linsen [74] increases $k$ until the maximum angle spanned by the point $x_i$ and two neighbors is below a threshold. In Figure 3.3, we illustrate the technique with $k = 8$, $k = 12$, and $k = 16$ on the same grid as in Figure 3.2. In all cases, the method produces ellipses with inconsistent orientation and aspect ratios, not reflecting the anisotropic distribution of the sampling positions.

**FIGURE 3.2** Local Gaussian filters for a jittered grid computed by covering the Voronoi cells: (a) visualization of the filters using a cutoff radius of $r = 0.25$, (b) $r = 0.5$.

**FIGURE 3.3** Gaussian filters computed using $k$-nearest neighbors: (a) $k = 8$, (b) $k = 12$, (c) $k = 16$. The cutoff radius is $r = 0.25$ in all cases.
3.1.3 Nonuniform Sampling

As introduced in the previous section, we represent a continuous signal \( g(x) \) reconstructed from nonuniform samples \( s_i \) at positions \( x_i \) as a weighted sum of reconstruction kernels (see also Equation 3.5):

\[
g(x) = \sum_i s_i h_i(x - x_i). \tag{3.11}
\]

Given a continuous, square integrable input signal \( f(x) \), the goal of nonuniform sampling is to determine the samples (or coefficients) \( s_i \) such that \( g \) approximates \( f \) as accurately as possible. In other words, the goal is to minimize the approximation error \( e \), which is measured using the \( L_2 \) norm of the difference between \( f \) and \( g \):

\[
e = \|g(x) - f(x)\|_{L_2}^2, \tag{3.12}
\]

where the \( L_2 \) norm is defined as \( \|f\|_{L_2}^2 = \int |f|^2 dx \).

This problem is well studied in the field of approximation theory. In particular, let us assume that the kernels \( h_i \) are orthonormal, i.e., for their inner products it holds that \( \langle h_i, h_j \rangle = \delta_{ij} \), where the inner product is defined as

\[
\langle f, g \rangle = \int f(x)g(x)dx \tag{3.13}
\]

and the Kronecker \( \delta_{ij} \) is

\[
\delta_{ij} = \begin{cases} 
0 & \text{for } i \neq j \\
1 & \text{for } i = j
\end{cases} \tag{3.14}
\]

Then, the coefficients \( s_i \) found by computing the inner products

\[
s_i = \langle f, h_i \rangle \tag{3.15}
\]

result in the minimum \( L_2 \) error in Equation 3.12.

As an example, we consider a set of 1D sinc functions \( h_i \), which are translated by integer values:

\[
h_i = \frac{\sin(\pi(x_i-x))}{\pi(x_i-x)}, \tag{3.16}
\]

where \( x_i = i \) and \( i \in \mathbb{Z} \). It is easy to verify that these functions are orthonormal, i.e., \( \langle h_i, h_j \rangle = \delta_{ij} \). Therefore, to approximate a square integrable function \( f \) by a weighted sum of these \( h_i \), we compute the coefficients \( s_i \) by evaluating the inner products

\[
s_i = \langle f, h_i \rangle = \int f(x)\frac{\sin(\pi(x_i-x))}{\pi(x_i-x)}dx. \tag{3.17}
\]

However, Equation 3.17 is equivalent to the convolution of \( f \) with an ideal low-pass filter \( h^{1/2} \) with cutoff frequency \( 1/2 \) (Equation 2.8), evaluated at the uniform positions \( x_i \):
\[ s_i = (f \otimes h^{1/2})(x_i), \text{ with } h^{1/2} = \frac{\sin(\pi x)}{\pi x}. \] 

(3.18)

Therefore, band-limiting a signal with an ideal low-pass filter and then evaluating the result at uniform sampling positions leads to a reconstruction of the sampled signal with minimum \( L_2 \) error.

Unfortunately, the Gaussian kernels that we compute in Section 3.1.2 are not orthogonal. Hence, evaluating inner products as in Equation 3.15 does not yield coefficients \( s_i \) with minimum \( L_2 \) error. In the following, we discuss three alternative sampling algorithms that are tailored for adaptive Gaussian kernels.

**Prefiltering.** Eventhough the Gaussian kernels are not orthonormal, it is a reasonable approach to determine the coefficients \( s_i \) by convolving the input signal \( f \) with the Gaussians \( h_i \) (as computed in Section 3.1.2) and evaluating at the nonuniform positions \( x_i \), i.e.,

\[
\begin{align*}
    s_i &= (f \otimes h_i)(x_i) = \int f(x) h_i(x_i - x) \, dx. \tag{3.19}
\end{align*}
\]

Clearly, this does not lead to a reconstruction with minimum \( L_2 \) error. However, Equation 3.19 represents a prefiltering approach, where the input signal \( f \) is band-limited before sampling. Because of the low-pass characteristics of the Gaussians, most aliasing artifacts in the reconstructed signal will be avoided (see also Figure 3.7).

Note that in our applications, the continuous input signal \( f(x) \) is actually often reconstructed from sampled data, too; usually, the input signal represents a texture image. Therefore, \( f(x) \) can be written as a weighted sum of samples \( f_j \) and reconstruction kernels \( r \) (which corresponds to the convolution of the sampled signal with the reconstruction filter):

\[
    f(x) = \sum_j f_j \cdot r(x - \xi_j), \tag{3.20}
\]

where the sampling positions \( \xi_j \) typically lie on a uniform grid. Substituting Equation 3.20 into Equation 3.19 yields

\[
\begin{align*}
    s_i &= \sum_j f_j \int r(x - \xi_j) h_i(x_i - x) \, dx = \sum_j f_j \int r(\tau) h_i(x_i - \xi_j - \tau) \, d\tau \\
    &= \sum_j f_j \cdot (r \otimes h_i)(x_i - \xi_j). \tag{3.21}
\end{align*}
\]

Here, the samples \( s_i \) are obtained as a weighted sum of reconstruction filters \( r \) that are convolved with the local low-pass filters \( h_i \). In fact, this is analogous to the resampling filters we derived in Section 2.7 (see Equation 2.37), except that we resample to a nonuniform grid using local low-pass filters. Using Gaussian kernels for all filters in Equation 3.21, i.e., \( r(x) = g_R(x) \) and \( h_i(x) = g_{H_i}(x) \) as in Section 2.7.4, we efficiently compute the samples as

\[
    s_i = \sum_j f_j \cdot g_{R \otimes H_i}(x_i - \xi_j). \tag{3.22}
\]
3.1 LOCAL FILTERING

**Optimized Sampling.** While the above approach is inspired by approximation theory, or by the procedure used for uniform sampling, respectively, it is a heuristic: although it is capable of avoiding most aliasing artifacts due to the local low-pass filters, it does not guarantee minimum $L_2$ error of the reconstructed signal. Next, we derive an algorithm to minimize the $L_2$ error under the assumption that sampled signals are reconstructed using a set of kernels that are not orthonormal, such as the Gaussian kernels derived in Section 3.1.2.

Let us denote the continuous function reconstructed from the nonuniform samples $s_i$ by $g(x)$. As in Equation 3.11, it has the form

$$g(x) = \sum s_i h_i(x - x_i), \quad (3.23)$$

where $h_i$ are the local filters as computed in Section 3.1.2, adapted to the local sampling density. Our goal now is to minimize the difference between the original signal $f(x)$ and the reconstructed signal $g(x)$. Utilizing the $L_2$ norm to measure the error, the problem is minimizing the following functional:

$$F(s) = \|g(x) - f(x)\|_{L_2}^2, \quad (3.24)$$

which, due to the form of $g(x)$ as in Equation 3.23, depends on the vector of samples $s = [s_i]$. Substituting Equation 3.23 into Equation 3.24 yields:

$$F(s) = \left\| \sum_i s_i h_i(x - x_i) - f(x) \right\|_{L_2}^2 = \langle \sum_i s_i h_i(x - x_i), \sum_i s_i h_i(x - x_i) \rangle + \langle f(x), f(x) \rangle - 2 \langle \sum_i s_i h_i(x - x_i), f(x) \rangle \quad (3.25)$$

We minimize $F(s)$ by computing the roots of its gradient, i.e.:

$$\nabla F(s) = \left[ \ldots, \frac{\partial F}{\partial s_i}, \ldots \right]^T = 0. \quad (3.26)$$

The partial derivatives $\frac{\partial F}{\partial s_i}$ are given by:

$$\frac{\partial F}{\partial s_i} = \frac{\partial}{\partial s_i} \sum_j s_j s_k \langle h_j(x - x_j), h_k(x - x_k) \rangle - 2 \frac{\partial}{\partial s_i} \langle \sum_i s_i h_i(x - x_i), f(x) \rangle$$

$$= 2 \sum_j s_j \langle h_j(x - x_j), h_i(x - x_i) \rangle - 2 \langle h_i(x - x_i), f(x) \rangle = 0 \quad (3.27)$$

This is a set of linear equations in the $s_i$ that can be written in matrix form as follows:
In fact, Equation 3.28 is a generalization of Equation 3.21: if the low-pass filters $h_i$ are orthonormal, i.e., $\langle h_i, h_j \rangle = \delta_{ij}$ (where $\delta_{ij}$ is the Kronecker delta), then $M$ is the identity matrix. Hence, the samples $s_i$ are given directly by the right hand side of Equation 3.28:

$$s_i = \langle h_i, f \rangle,$$

which is equivalent to Equation 3.15. In general, our low-pass filters $h_i$ are not orthogonal and $M$ is not a diagonal matrix, but it has a diagonal structure. In this case, Equation 3.19 can also be interpreted as an approximation to sampling with minimum $L_2$ error.

We next describe the implementation of Equation 3.28 assuming that the original continuous signal $f(x)$ is reconstructed from sampled data, i.e., that it has the form of Equation 3.20. Hence, the $i$-th entry $f_i$ of the vector $f$ in Equation 3.28 has the form

$$f_i = \sum_k f_k \cdot \langle h_i, r \rangle.$$

Further, Gaussian functions are chosen as low-pass filters $h_i$ and reconstruction kernels $r$, i.e., $h_i = g_{\mathbf{H}}$ and $r = g_{\mathbf{R}}$.

To evaluate the elements of the matrix $M$ and the vector $f$, we need to compute inner products of Gaussians that are translated to the sampling positions $x_s$ and $x_t$, respectively. Let us consider the inner product of two gaussians $g_V$ and $g_W$ with translations $s$ and $t$ respectively:

$$\langle g_V(x+s), g_W(x+t) \rangle = \int g_V(x+s) \cdot g_W(x+t) dx$$

$$= \int \frac{1}{4\pi^2 |V|^{1/2} |W|^{1/2}} e^{-\frac{1}{2}(x+s)^t V^{-1}(x+s) - \frac{1}{2}(x+t)^t W^{-1}(x+t)} dx$$

The exponent in Equation 3.31 has the general form

$$\frac{1}{2}(c_0 x_0^2 + c_1 x_0 x_1 + c_2 x_1^2 + c_3 x_0 + c_4 x_1 + c_5), \text{ where } x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}.$$
Scattered Data Interpolation. A third alternative to sample the continuous function \( f(x) \) is to enforce that the reconstructed function \( g(x) \) interpolates the original function at the sampling positions. More formally:

\[
g(x_k) = \sum_{i=0}^{n-1} s_i h_i(x_k - x_i) = f(x_k) \tag{3.36}
\]

for all sampling positions \( x_k, \ k = 0 \ldots n - 1 \). Here the \( h_j \) are the local reconstruction filters that are computed as described in Section 3.1.2. The linear relations of Equation 3.36 can be written in matrix form:
and the sample values $s_i$ are found by solving this linear equation system. This scheme is well known as scattered data interpolation using radial basis functions. We refer to the literature, e.g., [17, 30, 116], for more information about suitable kernels $h_i$, the numerical properties of the linear equation system, and efficient solving strategies.

**Comparison.** We compare the three sampling strategies described above using two experiments, illustrated in Figures 3.4 and 3.7. In the first experiment, we sampled the checkerboard texture shown in Figure 3.4a on the nonuniform grid shown in Figure 3.4b. The input texture is represented as a band-limited, continuous function. The continuous signal is actually reconstructed with Gaussian kernels from sampled data on a uniform grid (see Equation 3.20).

The sampling grid in Figure 3.4b was constructed by randomly jittering a uniform grid with 20 × 20 sampling points. On this nonuniform grid, we use Gaussian filters with identity variance with regard to the underlying uniform grid; i.e., the local filters $h_i$ (Equations 3.19, 3.23, and 3.36 for prefiltering, optimized sampling, and interpolating sampling, respectively) are unit Gaussians. Note that the orientation of the grid does not correspond to the orientation of the rows and columns in the original checkerboard texture.

EWA sampling is depicted in Figure 3.4c. The built-in low-pass filter significantly blurs the original function. The error, visualized in Figure 3.4d, reaches a maximum of $2.515 \times 10^{-1}$ at the sharp corners between black and white areas, whereas the original function is in the range $[0 \ldots 1]$; hence the relative error is more than 25%. The maximum error and the squared $L_2$ error over the whole area for the three sampling strategies are summarized in Table 3.1.

Optimized resampling, shown in Figure 3.4e, reproduces the original signal much more accurately, leading to a visually almost indistinguishable reconstruction. Its maximum error and the $L_2$ error over the whole area are summarized in Table 3.1. A visualization of the error is shown in Figure 3.4f on the right. We have not proven that the system matrix (see Equation 3.28) is guaranteed to be positive definite. Therefore, we use an iterative algorithm using symmetric LQ decomposition to solve the system of linear equations. In contrast to conjugate gradient approaches, this method does not require the system matrix to be positive definite. Unfortunately, the matrix is not well conditioned. In our experiment, it has a size of $400 \times 400$ elements and its condition number is $1.751 \times 10^7$. Hence the symmetric LQ solver converged very slowly, it needed 2904 iterations to reduce the norm of the residual $\varepsilon$ to $\varepsilon < 1 \times 10^{-4}$. Moreover, further increasing the number of iterations did not decrease the norm of the residual any more.

Note that due to the global support of the Gaussian filters, the system matrix is fully occupied, although most matrix elements are almost zero and the matrix has a strong diagonal structure. This is shown in Figure 3.5, where the upper left 50 × 50 matrix elements are visualized using a grayscale mapping.
In our experiment, the coefficients of the Gaussian filters resulting from the optimization procedure were in a range between $-55.2$ and $61.5$, although the sampled function has values in the interval $[0 \ldots 1]$. Therefore, to visualize the reconstructed continuous signal, we used Gaussians truncated to a rather large cutoff radius of $r = 4$ (see Equation 2.27). If the cutoff radius is chosen smaller, artifacts due to the truncation of the Gaussians to a finite support appear. This is illustrated in Figure 3.6, where the checkerboard texture is reconstructed using different cutoff radii.
Similar to the optimized sampling approach, sampling using scattered data interpolation leads to a very accurate reconstruction of the original signal, as shown in Figure 3.4g. Again, we used the symmetric LQ solver in our experiment, which converged to a residual norm of $\varepsilon < 1 \times 10^{-4}$ in 161 iterations. Convergence is faster since the system matrix has a condition number of $3.297 \times 10^3$, which is significantly lower than above. Although the matrix is fully occupied, too, its diagonal structure is stronger than with optimized sampling, as illustrated in Figure 3.5. The maximum error and the $L_2$ error for interpolating sampling are even smaller than the error resulting from optimized sampling, as shown in Table 3.1. We believe this is because the numerical solution of the equation systems is not precise enough.

In a second experiment, we sampled a larger region of the continuous checkerboard texture, shown in Figure 3.7a, to the same jittered, nonuniform sampling grid as above (Figure 3.7b). Here, the maximum frequency of the checkerboard texture is twice as high as in the first experiment. Due to the higher frequencies, the blurriness of EWA sampling (Figure 3.7c) is much more apparent than above. As summarized in Table 3.2, the maximum error and the $L_2$ error are larger, too. Optimized sampling reduces the $L_2$ error by one order of magnitude; also visually, the reconstructed texture

### TABLE 3.1
Comparison of the error of the three sampling strategies with the texture of Figure 3.4.

<table>
<thead>
<tr>
<th></th>
<th>EWA Sampling</th>
<th>Optimized Sampling</th>
<th>Interpolating Sampling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Error</td>
<td>$2.515 \times 10^{-1}$</td>
<td>$4.714 \times 10^{-2}$</td>
<td>$1.681 \times 10^{-2}$</td>
</tr>
<tr>
<td>Squared $L_2$ Error</td>
<td>$1.454 \times 10^2$</td>
<td>$1.204 \times 10^0$</td>
<td>$1.050 \times 10^{-1}$</td>
</tr>
</tbody>
</table>

### FIGURE 3.5
Visualization of the diagonal structure of the system matrices using the upper left $50 \times 50$ elements: (a) optimal sampling, the maximum value on the diagonal is $0.0896$; (b) interpolating sampling, the maximum value on the diagonal is $0.1592$.  

Similar to the optimized sampling approach, sampling using scattered data interpolation leads to a very accurate reconstruction of the original signal, as shown in Figure 3.4g. Again, we used the symmetric LQ solver in our experiment, which converged to a residual norm of $\varepsilon < 1 \times 10^{-4}$ in 161 iterations. Convergence is faster since the system matrix has a condition number of $3.297 \times 10^3$, which is significantly lower than above. Although the matrix is fully occupied, too, its diagonal structure is stronger than with optimized sampling, as illustrated in Figure 3.5. The maximum error and the $L_2$ error for interpolating sampling are even smaller than the error resulting from optimized sampling, as shown in Table 3.1. We believe this is because the numerical solution of the equation systems is not precise enough.

In a second experiment, we sampled a larger region of the continuous checkerboard texture, shown in Figure 3.7a, to the same jittered, nonuniform sampling grid as above (Figure 3.7b). Here, the maximum frequency of the checkerboard texture is twice as high as in the first experiment. Due to the higher frequencies, the blurriness of EWA sampling (Figure 3.7c) is much more apparent than above. As summarized in Table 3.2, the maximum error and the $L_2$ error are larger, too. Optimized sampling reduces the $L_2$ error by one order of magnitude; also visually, the reconstructed texture...
(Figure 3.7e) is much sharper. Interestingly, unlike in the first experiment, the error of interpolating sampling with the high frequency texture is larger than the error resulting from optimized sampling. The reason for this is aliasing: whereas optimized sampling minimizes the difference between the continuous input function and the continuously reconstructed output function, interpolating sampling works with a point sampled representation of the input function. However, point sampling the high frequency input function of Figure 3.7a on the grid shown in Figure 3.7b leads to aliasing. In the reconstructed signal shown in Figure 3.7g, aliasing artifacts appear as irregularities in the shape of the checkerboard boxes.

**Discussion.** Since Gaussian filters do not form an orthogonal basis for $L_2$ functions, neither on uniform nor on nonuniform grids, EWA sampling is not optimal in terms of the $L_2$ error in the reconstruction. On the other hand, EWA sampling avoids most aliasing artifacts because it inherently prefilters the input signal. By definition, optimized sampling should result in a minimal $L_2$ error. However, computing the optimized coefficients of Gaussian filters leads to a system of linear equations that is poorly conditioned. It cannot be solved robustly even for small sampling grids (i.e., containing more than a few hundreds of samples). Interpolating sampling leads to similar results as optimized sampling for low-frequency input signals. The corresponding linear equation system is even better conditioned and can be solved more accurately. However, interpolating sampling suffers from aliasing artifacts when sampling high frequency textures, since it uses a point sampled representation of the continuous input signal. Both optimized and interpolating sampling are not suitable for practical applications with several hundreds of thousands of samples, because the solution of the linear equation systems requires an unacceptable amount of computation time. There-
**FIGURE 3.7** Comparison of sampling strategies with a high frequency texture: (a) original continuous texture, (b) nonuniform sampling grid, (c) EWA sampling, (d) error of EWA sampling, (e) optimal sampling, (f) error of optimal sampling, (g) interpolating sampling, (h) error of interpolating sampling.

**TABLE 3.2** Comparison of the error of the three sampling strategies with the high frequency texture of Figure 3.7.

<table>
<thead>
<tr>
<th></th>
<th>EWA Sampling</th>
<th>Optimized Sampling</th>
<th>Interpolating Sampling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Error</td>
<td>$4.422 \times 10^{-1}$</td>
<td>$1.719 \times 10^{-1}$</td>
<td>$3.230 \times 10^{-1}$</td>
</tr>
<tr>
<td>Squared $L_2$ Error</td>
<td>$5.846 \times 10^{2}$</td>
<td>$2.277 \times 10^{1}$</td>
<td>$4.693 \times 10^{1}$</td>
</tr>
</tbody>
</table>
fore, we use EWA sampling to acquire models for point based rendering (see also Section 6.1).
In Chapter 2 and Chapter 3, we have discussed the reconstruction of smooth \textit{functions over a 2D domain} from uniformly and nonuniformly sampled data. In this chapter, we will extend some of these concepts to accomplish the smooth reconstruction of \textit{manifolds} embedded in a higher dimensional space. For our application, we focus on manifold surfaces that are embedded in three-dimensional Euclidean space. The \textit{surface samples} that we deal with contain all attributes of a surface at a certain point. However, they do not store any information about the surface topology, such as connectivity information in triangle meshes. The samples include the position of the surface, i.e., information about the embedding of the 2D surface in 3D, as well as samples of additional functions that are defined on the surface. For graphics objects, these functions specify the visual appearance, or the light reflectance properties, of the surface. They represent, e.g., the diffuse color, the specular color, the glossiness, or even higher dimensional quantities such as BRDFs [80]. Our method allows the reconstruction of the surface position and its attribute functions directly from the point samples in a unified framework.

In Section 4.1, we summarize related work in the context of surface reconstruction from unstructured point samples, and then in Section 4.2 introduce our method that is based on a \textit{parameterization} of the points. Two applications of our surface representation, namely rendering and texture mapping, will be presented in Chapter 5 and Chapter 8 respectively.

\section{4.1 RELATED WORK}

Mathematical representations for two dimensional surfaces in three dimensional space can be classified into two groups, both with their specific advantages for computer graphics applications. On the one hand, \textit{implicit surfaces} are defined as the zero set of a scalar function over the whole three dimensional domain. These representations have
a particularly simple algebraic structure, which is largely independent of the topological complexity of the surface. Surface deformations and topological changes can be applied easily while the global consistency of the surface is guaranteed by construction. Popular choices to specify the scalar field are radial basis functions [20] or level sets [96]. Point set surfaces [3] are based on moving least squares approximation and a projection operator to implicitly define the surface. This representation is similar to ours in that it directly uses the unstructured point cloud to define a smooth surface. We provide a conceptual comparison of moving least squares approximation to our approach in Section 4.3.

On the other hand, parametric surfaces are defined by a mapping of a two-dimensional domain into three-dimensional space. Among the most popular parametric surfaces in computer graphics are Bézier, B-spline, NURBS, and subdivision surfaces. For a survey see for example Farin’s book [34]. These representations are based on a mesh of control points with known connectivity. Many representations, such as B-spline and NURBS surfaces, require the mesh to have a regular connectivity, while others, such as subdivision surfaces, work with semi-regular meshes. In contrast, our approach does not need any connectivity information or mesh structure. Rather, the surface is given directly by the unstructured, parameterized point cloud. While certain modeling operations affecting the surface topology are more complicated to handle with parametric surfaces, they are more suitable for rendering and texture mapping. We discuss these applications with our surface representation in Chapter 5 and Chapter 8 respectively.

Most parametric surface representations are based on the encoding of the surface topology in a triangular or quadrilateral mesh. Therefore, such a mesh usually has to be reconstructed from surface samples in a pre-processing step. There is a multitude of sophisticated algorithms that achieve this task. Popular approaches use volumetric distance functions and isosurface extraction, e.g., [53, 25], while others are based on Delaunay triangulations [32, 6, 41]. These contributions also include theoretical results that provide sampling criteria specifying whether the correct reconstruction of the surface topology is possible from a given point cloud. As a major advantage, our surface representation works with an unstructured point cloud without connectivity information. Therefore, we do not need to reconstruct the surface topology and build up a mesh structure.

4.2 SMOOTH SURFACE RECONSTRUCTION FROM PARAMETERIZED POINT SAMPLES

In this section, we introduce a parametric surface representation that is defined directly by an unstructured set of surface samples. The sample points are neither required to lie on a uniform grid, nor does any connectivity information between the samples have to be known. Conceptually, the construction of the surface proceeds in two steps: In the first step, we locally approximate the surface and all its attribute functions in a small neighborhood around each sample point. This requires the computation of local parameterizations in the neighborhood of each point. In the second step, we blend the local approximations using a global parameterization of the surface. For this purpose, we establish a mapping of each local parameterization into the global parameter domain, which therefore serves as a common frame of reference for the local surface approxi-
mations. To perform blending, we build a weighted sum of the local approximations multiplied by local reconstruction kernels, similar as in Section 3.1.

More formally, a surface sample \( S \) is given as a set of samples of the surface attributes at a certain point, including surface position \( \mathbf{p} \), color \( \mathbf{c} \), etc., i.e., \( S = \{ \mathbf{p}, \mathbf{c}, \ldots \} \). The surface position \( \mathbf{p} \) has a special role, since it is used to compute the local parameterizations around each sample \( S_i \). Let us denote the coordinates of the local parameterization around \( S_i \) by \( \mathbf{u}_i = (u_{i,0}, u_{i,1}) \) and let \( N_i \) be the index set of the \( k \)-nearest neighbors of \( \mathbf{p}_i \). We determine the local coordinates \( \mathbf{u}_j^i \) of a sample \( S_j \), \( j \in N_i \) by orthogonally projecting its position \( \mathbf{p}_j \) onto a reference plane \( e_i(\mathbf{u}_i) \). The reference plane is typically chosen such that it runs through \( \mathbf{p}_i \) and such that it has the smallest sum of the squared distances to the points \( \{ \mathbf{p}_j | j \in N_i \} \). We represent the reference plane by \( \mathbf{e}_i = (\mathbf{v}_{i,0}, \mathbf{v}_{i,1}) \), where \( \mathbf{v}_{i,0} \) and \( \mathbf{v}_{i,1} \) are two orthogonal unit vectors. Note that \( e_i(\mathbf{u}_i) = \mathbf{p}_i + u_{i,0} \mathbf{v}_{i,0} + u_{i,1} \mathbf{v}_{i,1} \), where \( \mathbf{v}_{i,0} \) and \( \mathbf{v}_{i,1} \) are two orthogonal unit vectors.

Then, for each attribute \( A \in S \) of the surface, we compute a smooth reconstruction that approximates the sampled values \( A_j \), \( j \in \{ N_i \cup i \} \) in the local parameter domain. For this purpose, we fit a low order polynomial \( p_i^A(\mathbf{u}_j^i) \), such that

\[
\sum_{j \in \{ N_i \cup i \}} (p_i^A(\mathbf{u}_j^i) - A_j)^2 = \min.
\]

This approach to compute local polynomial fitting functions of a surface is similar to the scheme proposed by [133]. However, we do not only compute the surface position \( \mathbf{p} \) in this way, but all surface attributes \( A \in S \).

Note that the attribute samples \( A_j^i \) may either be represented by scalar values, e.g., for the surface glossiness, such that \( p_i^A : \mathbb{R}^2 \rightarrow \mathbb{R} \), or by vectors, e.g., for the surface position, such that \( p_i^A : \mathbb{R}^2 \rightarrow \mathbb{R}^3 \). To simplify the process in practice, we use constant functions \( p_i^A(\mathbf{u}_j^i) = A_i^j \) for all attributes except the surface position, where we fit a linear polynomial. Also, if the surface normals are given explicitly at each point, we may use the tangent plane instead of a least squares fitting plane as the reference for the local parameterization and as the local linear approximation of the surface position.

We further compute local reconstruction kernels \( r_i^j(\mathbf{u}_i) \) that will be used to blend the local fits. The kernels can be interpreted as a weight indicating the confidence that the local fitting functions accurately represent the surface attributes. Currently, we use Gaussian kernels that are determined as described in Section 3.1.2, whereas the non-uniform 2D sampling grid is given by the local coordinates \( \mathbf{u}_j^i \) of the samples \( S_j \), \( j \in \{ N_i \cup i \} \).

The core idea to combine the local, smooth approximations \( p_i^A \) is to represent them in a common frame of reference. This is achieved by specifying invertible 2D to 2D mappings \( \varphi_i^j : \mathbf{u}_i \rightarrow \mathbf{x} \) that transform local coordinates \( \mathbf{u}_i \) into global coordinates \( \mathbf{x} \). Let us denote the reconstruction kernels and the fitting functions mapped to the global parameter domain by

\[
r_i^j(\mathbf{x}) = r_i^j(\varphi_i^{-1}(\mathbf{x})) \quad \text{and} \quad p_i^A(\mathbf{x}) = p_i^A(\varphi_i^{-1}(\mathbf{x})).
\]

The global reconstruction \( s_A(\mathbf{x}) \) of a surface attribute \( A \), parameterized by the global coordinates \( \mathbf{x} \), is then computed by building the sum
similar to adaptive weighted average filtering as described in Section 3.1.2. Therefore, we call this approach surface reconstruction using parameterized, adaptive weighted average filtering. A typical scenario is to use linear mapping functions $\varphi_i$, Gaussian kernels $r_i$, and constant or linear fitting functions $p_i^p$, which are all $C^\infty$-continuous. Note that in this case, the function $s_A(x)$ represented by Equation 4.3 is $C^\infty$-continuous, too.

We illustrate Equation 4.3 in Figure 4.1, where we reconstructed a 2D curve $s_p(x)$ from sampled positions $p_i$, $i = 1 \ldots 8$. To each point $i$ we assigned global parameter coordinates $x_i$:

$$x_i = \sum_{j=1}^{i} \|p_j - p_{j-1}\| \text{ for } i > 1 \text{ and } x_1 = 0.$$  \hfill (4.4)

We chose the fitting functions $p_i^p(u_i)$ to correspond to the least squares plane through $p_i$. We parameterized them by arc length, i.e.,

$$\|p_i^p(u_i^1) - p_i^p(u_i^0)\| = \|u_i^1 - u_i^0\|$$  \hfill (4.5)

for any values $u_i^0$ and $u_i^1$, and by setting $p_i^p(0) = p_i$. The reconstruction kernels $r_i(u_i)$ were Gaussians with variance

$$\sigma_i = |x_{i+1} - x_{i-1}|/2.$$  \hfill (4.6)

Finally, we mapped global to local parameters using $\varphi_i^{-1}(x) = x - x_i$. Obviously, the curve $s_p(x)$ shown in Figure 4.1a does not interpolate the samples $p_i$. On the other hand, the curve is $C^\infty$-continuous, since $p_i^p$, $r_i$, and $\varphi_i^{-1}$ are $C^\infty$-continuous. Finally, Figure 4.1b shows the Gaussian reconstruction kernels $r_i(x)$ in the global parameter domain.

In this thesis, we will present two applications using surfaces represented by Equation 4.3, which differ in the way the mapping from local to global parameters is computed: In Chapter 5, we introduce an efficient rendering algorithm that uses the projection plane as the global parameter domain and reconstructs the continuous surface directly in image space. In Chapter 8, we discuss an interactive texture mapping application that computes the global parameterization from user defined feature points.

The choice of appropriate reconstruction kernels $r_i(u_i)$ has a significant influence on the quality of the surface reconstruction, as we illustrate in Figure 4.2. While Figure 4.2b visualizes the global parameterization of a 3D surface patch using isoparameter lines, Figures 4.2a and 4.2c show the support of the reconstruction kernels in the corresponding 2D parameter domain for two different choices of kernels. In Figure 4.2a, we have computed the reconstruction kernels adaptively as described in Section 3.1.2, whereas the kernels in Figure 4.2c are chosen independently of the local sampling density. In Figures 4.2d and 4.2e, the kernels are visualized on the local ref-
4.2 SMOOTH SURFACE RECONSTRUCTION FROM PARAMETERIZED POINT SAMPLES

erence planes, and Figures 4.2f and 4.2g show the reconstructed surfaces. Because of the constant reconstruction kernels used in Figure 4.2g, the fine detail in the densely sampled areas is lost. On the other hand, reconstruction with adaptive filters results in a smooth and highly detailed surface (Figure 4.2f).

Comparing Figures 4.2f and 4.2g, it is obvious that the reconstruction filters have a band-limiting effect on the reconstructed surface. Wide filters in the spatial domain suppress high frequencies in the signal, as can be seen in the center region of Figure 4.2f. Following signal processing theory (Chapter 2), band-limiting the surface is achieved by convolving \( s_A(x) \) with a low-pass filter kernels \( h(x) \). We approximate the convolution by only applying it to the reconstruction kernels \( r_i'(x) \) in the global parameter domain:

\[
(s_A \otimes h)(x) \approx \frac{\sum (r_i' \otimes h)(x) \cdot p_i^A(x)}{\sum (r_i' \otimes h)(x)}. \tag{4.7}
\]

For constant fitting functions, i.e., \( p_i^A = A_i \), this approximation is exact except for the normalization factor. Note that the band-limited reconstruction filter \( r_i' \otimes h \) has the form of a resampling filter in the global parameter domain, as presented in Section 2.7. Hence by choosing the low-pass filter \( h \) appropriately, we can avoid aliasing artifacts during resampling of the surface on a uniform grid in the global parameter domain. This technique is applied in our rendering algorithm described in Chapter 5.

Note that this approach of filtering the surface in the parameter domain (parametric fairing) usually leads to shrinkage effects. Smoothing the three spatial coordinates of the surface separately reduces each component to a constant, which means that the surface collapses to a single point. However, this effect can be avoided when we use non-constant fitting functions, e.g., linear functions. In Figure 4.3, we compare curve smoothing with linear fitting functions to parametric smoothing (i.e., constant fitting functions). While Figure 4.3a shows the original, noisy curve, Figure 4.3b depicts the curve after smoothing with linear fitting functions, where a circle with the original

**FIGURE 4.1** Curve reconstruction using parameterized, adaptive weighted average filtering: (a) reconstructed curve \( s_p(x) \), (b) global parameter domain with Gaussian reconstruction filters.
radius is restored. On the other hand, Figure 4.3c shows the curve after parametric smoothing, where the radius of the circle is reduced by the smoothing process. In both cases, we used a Gaussian kernel with a variance corresponding to one tenth of the circle circumference.

We have applied this smoothing technique to surfaces, too, as illustrated in Figure 4.4. Figure 4.4a depicts the surface samples, from which we reconstructed the continuous surface shown in Figure 4.4b. In the neighborhood visualized in Figure 4.4c, we convolved the surface reconstruction kernels with a low-pass filter as in Equation 4.7, resulting in the locally smoothed surface depicted in Figure 4.4d.
4.2 SMOOTH SURFACE RECONSTRUCTION FROM PARAMETERIZED POINT SAMPLES

FIGURE 4.3 Curve smoothing: (a) original curve with noise, (b) smoothing with linear fitting functions avoids shrinkage, (c) parametric smoothing leads to shrinkage.

FIGURE 4.4 Surface smoothing: (a) surface samples, (b) continuous surface reconstruction, (c) surface area that is subject to smoothing, (d) locally smoothed surface.
4.3 COMPARISON TO MOVING LEAST SQUARES APPROXIMATION

The moving least squares (MLS) technique is an alternative method to reconstruct, or approximate, continuous functions and surfaces from nonuniform samples that shares many similarities with our approach presented in Section 4.2. MLS has been introduced in the functional setting by Lancaster and Salkauskas [62] and extended to the reconstruction of manifold surfaces by Levin [67]. To computer graphics, it has been introduced by Alexa et al. [3]. MLS is similar to our method in that it uses polynomial fitting functions in combination with weight functions to compute smooth reconstructions. However, the procedure how the fitting functions are computed and the way the weight functions are applied is rather different. In this section, we compare the two approaches on a conceptual level without introducing any formalism for describing MLS, pointing out the fundamental differences between them.

We compare the two approaches using a 1D example in the functional setting illustrated in Figure 4.5. Figures 4.5a and 4.5b visualize our approach described in Section 4.2. In Figure 4.5a, ten linear fitting functions are depicted, each computed as a linear least squares polynomial through a center point and its left and right neighbors. Center points and their associated fitting polynomials have the same shade of gray. Figure 4.5b shows the corresponding Gaussian reconstruction kernels. In our approach, smooth reconstruction is performed by building the normalized weighted sum of all fitting polynomials and reconstruction kernels as given by Equation 4.3.

Reconstructing a smooth function using MLS is illustrated in Figures 4.5c and 4.5d. To evaluate the MLS function at a given position \( x \), a least squares polynomial is computed at this position that minimizes the error with respect to all input points. However, the error induced by each input point is multiplied with the reconstruction, or weighting kernel, centered at \( x \). Because the weighting function moves with the evaluation position \( x \), the resulting polynomial is called the moving least squares (MLS) polynomial. The reconstructed function value is then given by evaluating the MLS polynomial at \( x \). In Figures 4.5c and 4.5d, we show the weighting kernel and the linear MLS polynomial at \( x = 0.9 \) and \( x = 1.8 \) respectively.

While in our approach the reconstruction kernels are attached to the input points, the MLS weighting kernels are centered at the position where the reconstructed function is evaluated. Further, in our approach the reconstruction kernels are used as weights for blending the fitting polynomials, while in MLS the weighting kernels determine the influence of the error of each input point when computing the least squares polynomial. In our approach, the smoothness of the reconstructed function does not depend on the way the fitting polynomials are computed. For example, we can use all input points or only \( k \)-nearest neighbors of a center point. However, to ensure a certain smoothness of the MLS reconstruction, all input points that lie in the support of the weighting kernel have to be considered.

Finally, the extension of MLS from the functional setting to the reconstruction of manifold surfaces has been achieved by the introduction of a nonlinear projection operator [67]. Hence, MLS surfaces are defined implicitly as those points in \( \mathbb{R}^3 \) that project onto themselves. On the other hand, our approach generalizes to the reconstruction of surfaces by introducing a common frame of reference, i.e., a global parameterization, for all fitting polynomials and reconstruction kernels.
4.3 Comparison to Moving Least Squares Approximation

Figure 4.6 shows the reconstructed functions obtained from ten input points using our approach (Figure 4.6a) and MLS (Figure 4.6b). In both cases, we used linear least squares polynomials and Gaussian reconstruction, or weighting, kernels. While the variance of the Gaussian was $\sigma^2 = 0.1$ in our approach, it was $\sigma^2 = 0.25$ for MLS.

Figure 4.6 shows the reconstructed functions obtained from ten input points using our approach (Figure 4.6a) and MLS (Figure 4.6b). In both cases, we used linear least squares polynomials and Gaussian reconstruction, or weighting, kernels. While the variance of the Gaussian was $\sigma^2 = 0.1$ in our approach, it was $\sigma^2 = 0.25$ for MLS.

Figure 4.5 Comparison to MLS reconstruction: (a) fitting functions, (b) weighting kernels of our approach. (c) MLS evaluation at $x = 0.9$, (d) MLS evaluation at $x = 1.8$.

FIGURE 4.5 Comparison to MLS reconstruction: (a) fitting functions, (b) weighting kernels of our approach. (c) MLS evaluation at $x = 0.9$, (d) MLS evaluation at $x = 1.8$.

FIGURE 4.6 Reconstruction using our approach (a) and MLS (b).
CHAPTER 5

EWA SURFACE SPLATTING

In this chapter, we bring together the concepts of antialiasing using resampling filters presented in Chapter 2 and the parametric surface representation introduced in Chapter 4 to derive an efficient, high quality rendering algorithm for point-based graphics objects. The technique is based on elliptical Gaussian filters, hence we use the abbreviation EWA, standing for elliptical weighted average as introduced by Greene and Heckbert [44]. The term splatting refers to the fact that the resampling filters are projected to image space; therefore we also call these filters EWA splats. We have applied the concept of EWA splatting to point-sampled surface and volume data: EWA surface splatting for antialiased rendering of surfaces and EWA volume splatting, which is presented in Chapter 7.

EWA surface splatting is based on the parametrized surface representation introduced in Section 4.2. As input, it takes an unstructured set of surface samples \( \{ S_i \} \). Each sample \( S_i \) consists of a number of surface attributes, including surface position \( p_i \), diffuse color \( c_i \), and others, as well as the corresponding fitting functions, i.e., \( p_i^p(u_i), p_i^c(u_i) \), given in local coordinates \( u_i = (u_{i,0}, u_{i,1})^T \). For surface splatting, we choose all fitting functions to be constants, that is \( p_i = c_i \), etc., except for the surface position, which is locally approximated by a linear polynomial \( p_i^p \). Note further that this fitting function is set to coincide with the reference plane \( e_j \), i.e., \( p_i^p(u_i) = e_j(u_i) \) (see Section 4.2). If the surface samples include the surface normal \( n_j \), we determine \( e_j \) to match the corresponding tangent plane. Otherwise, we compute least squares fitting planes as described in Section 4.2. Additionally, each surface sample also has an associated local Gaussian reconstruction filter \( r_i(u_i) \) (see also Section 4.2).

Given this input data, the goal of surface splatting is to reconstruct a continuous surface in image space using Equation 4.3, and then to sample the surface on a uniform grid to generate the output raster image. To achieve this, we project the local fitting functions and reconstruction kernels to image space, i.e., the coordinate system of the output image. Then we use image space as the global parameter domain to blend the
local surface approximations and reconstruct a smooth surface. The 2D to 2D mappings that transform local coordinates to output image coordinates are at the core of our algorithm. These mappings include all the coordinate system transformations that commonly occur in 3D rendering pipelines (e.g., OpenGL [143]).

Furthermore, surface splatting applies a low-pass filter as a prefilter (see Section 2.4) before sampling the surface in image space, hence avoiding aliasing artifacts and providing high image quality.

In Section 5.1, we in detail explain the derivation of the mappings from local parameters to image space. We use the mappings to define Gaussian resampling filters in image space in Section 5.2. In Section 5.3 we provide more details about the surface splatting algorithm, and we further present a version of the algorithm that is suitable for hardware acceleration in Section 5.4. We conclude the chapter with the presentation of results and a comparison to other antialiasing techniques in Section 5.5.

5.1 CONTINUOUS SURFACE RECONSTRUCTION IN IMAGE SPACE

Let us denote image space coordinates by $\mathbf{x} = (x_0, x_1)^T$. As a key component of the surface splatting algorithm, we introduce the mappings $\Pi_i : \mathbb{R}^2 \rightarrow \mathbb{R}^2$, which transform local coordinates $\mathbf{u}_i$ to image space coordinates $\mathbf{x}_i$. Essentially, the $\Pi_i$ represent the projection of the reference planes $e_i(u_i) = p_{i,0} + u_{i,0}v_{i,0} + u_{i,1}v_{i,1}$ (see Section 4.2) to image space. The mappings consist of a concatenation of all the coordinate system transformations that are involved in rendering the surface: the 2D to 3D parameterization of the reference plane, the 3D to 3D viewing transformation, the 3D to 2D projection to the image plane, and the 2D to 2D viewport transformation to output image coordinates, as illustrated in Figure 5.1. Note that in the following, we ignore the 3D viewing and the 2D viewport transformations to simplify the explanations. These are usually affine transformations that are straightforward to include in the surface splatting algorithm. We assume that 3D input coordinates are given in camera space and perspective projection yields coordinates of the 2D output image.

Because of the perspective projection involved, the mappings $\Pi_i$ are non-linear, which makes it impossible to directly derive Gaussian resampling filters for surface rendering. Therefore we replace the $\Pi_i$ by their local linear approximations $\bar{\Pi}_i$, which are given by the first two terms of the Taylor expansion:
5.1 Continuous Surface Reconstruction in Image Space

\[ \overline{\Pi}_i(u_i) = x_i + J_i \cdot u_i, \quad (5.1) \]

where \( x_i = \Pi_i(u_i^i) = \Pi_i(0, 0) \) and \( J_i \) is the Jacobian of \( \Pi_i \) at \( u_i^i = (0, 0) \), i.e.,

\[ J_i = \frac{\partial \Pi_i}{\partial u_i}(0, 0) = \begin{bmatrix} \frac{\partial x_0}{\partial u_{i,0}} & \frac{\partial x_0}{\partial u_{i,1}} \\ \frac{\partial x_1}{\partial u_{i,0}} & \frac{\partial x_1}{\partial u_{i,1}} \end{bmatrix}. \quad (5.2) \]

In practice, we approximate the partial derivatives in Equation 5.2 using finite differences:

\[ \frac{\partial x_0}{\partial u_{i,0}} \approx \frac{\Pi_i(h, 0) - \Pi_i(0, 0)}{h}, \quad \text{and} \quad \frac{\partial x_1}{\partial u_{i,1}} \approx \frac{\Pi_i(0, h) - \Pi_i(0, 0)}{h}. \quad (5.3) \]

These finite differences are computed by projecting the vectors \( v_{i,0} \) and \( v_{i,1} \) spanning the reference plane \( e_i \) from camera to image space, as illustrated in Figure 5.2. Camera space has its origin at the center of projection and the image plane is perpendicular to one of the coordinate axes.

**FIGURE 5.2** The mappings \( \Pi_i \) transform local parameters \( u_i \) to image space coordinates \( x_i \).
With the mappings $\Pi_i$ at hand, the continuous surface in image space is given by

$$s_A(x) = \frac{\sum r'_i(x - x_i) \cdot p^A_i(x - x_i)}{\sum r'_i(x - x_i)},$$  \hspace{1cm} (5.4)

where $r'_i(x) = r_i(J_i^{-1} \cdot x)$ and $p^A_i(x) = p_i^A(J_i^{-1} \cdot x)$,  \hspace{1cm} (5.5)

for all surface attributes $A \in \{p, c, \ldots\}$, which is analogous to Equations 4.2 and 4.3. Here, $r'_i(x)$ and $p^A_i(x)$ can be interpreted as reconstruction filters and fitting functions that are projected to image space.

Further, we band-limit the surface function to the Nyquist limit of the output sampling grid by convolving it with a low-pass filter $h$, similar to Equation 4.7:

$$\left( s_A \otimes h \right)(x) \approx \frac{\sum_i (r'_i \otimes h)(x - x_i) \cdot p^A_i(x - x_i)}{\sum_i (r'_i \otimes h)(x - x_i)}.$$  \hspace{1cm} (5.6)

In the next section, we will describe how to implement Equation 5.6 with Gaussian reconstruction and low-pass filters, which can be combined into a Gaussian resampling filter.

Note that since the reconstructed surface position $s_p(x)$ depends on the mappings $\Pi_i$, the shape of the surface slightly varies for different viewpoints. However, these changes are not visible in practice. In the above explanations, we have also ignored the fact that some parts of the surface may be invisible from a given viewpoint, hence the corresponding fitting functions should not be accumulated in Equation 5.4 or 5.6, respectively. We address this issue in Section 5.3.

### 5.2 SURFACE RESAMPLING WITH GAUSSIAN FILTERS

For surface splatting, we have chosen Gaussian reconstruction filters $r_i(u_i) = g_R(u_i)$ and a Gaussian low-pass filter $h(x) = g_H(x)$ because of their useful analytical properties (see Section 2.6). While the variance matrices of the reconstruction filters are determined as described in Section 4.2, the low-pass filter usually has unit variance with respect to image space coordinates, hence $H = I$.

Using the results from Section 2.6, it is possible to express the Gaussian reconstruction filter mapped to image space as a Gaussian filter again:

$$r'_i(x) = r_i(J_i^{-1} \cdot x) = \frac{1}{|J_i^{-1}|} g_{J_i R_i J_i}(x),$$  \hspace{1cm} (5.7)

which is analogous to Equation 2.50 in Section 2.7.4. Following Equation 2.51, convolving $r'_i$ with the Gaussian low-pass filter $g_H$ yields a Gaussian resampling filter expressed in image space:
5.3 THE SURFACE SPLATTING ALGORITHM

To display the surface on a raster image, we evaluate Equation 5.9 on a uniform pixel grid. The surface splatting algorithm proceeds in two passes, as summarized in Figure 5.3.

\[
\rho_{i}(x) = (r'_{i} \otimes h)(x - x_{i})
\]

\[
= \frac{1}{|J_{i}^{-1}|}(g_{J_{i}R, J_{i}^{T}} \otimes g_{H})(x - x_{i})
\]

\[
= \frac{1}{|J_{i}^{-1}|}g_{J_{i}R, J_{i}^{T} + H}(x - x_{i})
\]  

(5.8)

Note that this formulation corresponds to the Gaussian destination domain resampling filter introduced in Section 2.7.4, using a warped reconstruction filter. With the \( \rho_{i} \), the rendered, band-limited surface in image space can be expressed as

\[
(s_{d} \otimes h)(x) \approx \frac{\sum_{i} \rho_{i}(x) \cdot p_{A}^{l}(x - x_{i})}{\sum_{i} \rho_{i}(x)}.
\]  

(5.9)

5.3 THE SURFACE SPLATTING ALGORITHM

FIGURE 5.3 The surface splatting algorithm.

In the first pass, we splat each surface sample separately into a framebuffer using the resampling filter of Equation 5.8 (line 2). Typically, the framebuffer stores the surface position and the shaded surface color only, but we could sample any other surface attribute in the framebuffer as well. For each sample, we compute the shaded surface color (line 3) by evaluating any illumination model, such as the standard Phong model, or surface reflectance fields [80], etc. All the required surface parameters (glossiness, etc., depending on the illumination model) need to be available as a surface attribute \( p_{A}^{l} \) of the sample \( S_{i} \). Next we compute the linear fitting polynomial \( p_{i}^{p} \) for the surface position (line 4). This function is used to determine the visibility of the surface sample by means of an extended z-buffer (Section 5.3.2). To compute Equation 5.9, the products of the resampling filter and the surface attributes are splatted, i.e., evaluated and
accumulated, in the pixels of the framebuffer (lines 5 and 6). Evaluating the resampling filter is described in detail in Section 5.3.1. Finally, the resampling filter itself is splatted, too, to serve as the normalization factor (line 7).

In the second pass, we normalize the values in each pixel by dividing through the accumulated weights of the resampling filters (line 10). As an alternative, we could also perform the shading operations on a per-pixel basis in the second pass, instead of shading per-sample as described above. The advantage of this approach is that unnecessary shading operations of occluded samples are avoided. On the other hand, this method requires that all surface attributes needed for shading are stored in the framebuffer.

5.3.1 Splatting

Splatting is the process of evaluating and accumulating the product of the resampling filters and the fitting functions at the output pixels in the framebuffer. Evaluating the linear fitting functions for the surface position \( p^R_i(x - x_i) \) is elaborated further below. Here, we explain how to compute the Gaussian resampling filters by incrementally rasterizing them in an axis aligned rectangular bounding box.

First, we determine the bounding box such that it encloses the elliptical area where the absolute value of the exponent \( q(x) \) of the Gaussian resampling filter is smaller than a given threshold:

\[
q(x) = \frac{1}{2}(x - x_i)^T (J_i R_i J_i^T + H)^{-1} (x - x_i) < \frac{1}{2} r^2, \tag{5.10}
\]

where \( r \) is the cutoff radius. Typically, \( r \) is chosen in the range \( 1 < r < 2 \) for surface splatting. Larger values for \( r \) lead to smoother reconstructions, but can also cause visual artifacts, as is shown in Section 5.5. We find the extents \([x_{0, min}, x_{0, max}]\) and \([x_{1, min}, x_{1, max}]\) of the bounding box by solving the following quadratic equations (see also Figure 5.4):

\[
\begin{align*}
  x_{0, min} & = 0, \quad q(x) = \frac{1}{2} r^2 \\
  x_{0, max} & = 0, \quad q(x) = \frac{1}{2} r^2 \\
  x_{1, min} & = 0, \quad q(x) = \frac{1}{2} r^2 \\
  x_{1, max} & = 0, \quad q(x) = \frac{1}{2} r^2
\end{align*} \tag{5.11}
\]

We evaluate the Gaussian resampling filter by incrementally rasterizing its exponent \( q(x) \) within the bounding box. Let us denote the matrix elements of the inverse variance matrix \( V^{-1} \) of the Gaussian by

\[
V^{-1} = (J_i R_i J_i^T + H)^{-1} = \begin{bmatrix} A & B/2 \\ B/2 & C \end{bmatrix}. \tag{5.12}
\]

Pseudo-code for rasterizing \( q(x) \) is given in Figure 5.5. Here, \( W(x) = \sum \rho_i(x) \) is the the factor that is used in the normalization pass (Figure 5.3).
5.3.2 Z-Buffering

In general, the depth complexity of a scene is greater than one. This means that the parameterization-by-projection approach introduced in Section 5.1 will merge different surface layers of an object into one. However, at each pixel with coordinates \( (x_0, x_1) \), only the visible surface samples should be accumulated in the framebuffer, while the others should be discarded. To achieve this, we use a z-buffer that contains camera space z-coordinates at each pixel. Camera space has its origin at the viewpoint, and its z-axis is perpendicular to the image plane, positive z-values lying in front of the viewpoint.

To simplify the explanations below, let us assume that the reference planes \( e_i \) (which correspond to the fitting functions for the surface position \( p_i^P \)) are expressed in camera coordinates, i.e., \( e_i(\mathbf{u}_i) = p_i^P(\mathbf{u}_i) = \mathbf{p}_i + u_{i,0} \mathbf{v}_{i,0} + u_{i,1} \mathbf{v}_{i,1} \), where \( \mathbf{p}_i \) is a point in camera space and \( \mathbf{v}_{i,0} \) and \( \mathbf{v}_{i,1} \) are camera space vectors. If \( p_i^P \) is given in a different coordinate system, it is easy to transform it to camera coordinates by transforming the 3D point \( \mathbf{p}_i \) and the vectors \( \mathbf{v}_{i,0} \) and \( \mathbf{v}_{i,1} \). Hence the surface position at a pixel \( k \) with coordinates \( \mathbf{x}_k \) is approximated by
which is a linear polynomial that can be rasterized efficiently.

In its simplest form, at each pixel the \( z \)-buffer stores the camera space \( z \)-coordinate of the fitting function \( p^P_i \) that is closest to the viewer. Given a \( z \)-value \( z(x_i) \) stored in pixel \( k \), and a new contribution of sample \( S_i \) with \( z \)-value \( p^P_i(x_k - x_i) \), one of the following three alternatives is chosen:

\[
\begin{align*}
\text{If } |p^P_i(x_k - x_i) - z(x_i)| < \epsilon, & \quad \text{the contributions of the surface sample are added to the data already stored in the framebuffer pixel.} \\
\text{If } p^P_i(x_k - x_i) - z(x_k) > \epsilon, & \quad \text{the new contributions are discarded.} \\
\text{If } p^P_i(x_k - x_i) - z(x_k) < -\epsilon, & \quad \text{the data stored in the framebuffer is replaced by the new contributions.}
\end{align*}
\]

This scheme is sometimes also called \( \epsilon \)-\( z \)-buffering. The basic approach described above can easily be extended towards a multi-layered framebuffer that can handle semi-transparent surfaces as well. Our method is inspired by the \( \mathbb{Z}^3 \)-algorithm [56], that provides order independent transparency using a single rendering pass and a fixed amount of frame buffer memory. In a multi-layered framebuffer, each layer stores a so called fragment at each pixel. The purpose of a fragment is to collect the contributions of a single surface to the pixel. After all samples have been splatted, the fragments are blended back-to-front to produce the final pixel color.

Of course, the number of different surfaces that potentially contribute to a pixel is unbounded. In order to handle this situation correctly, \( A \)-buffer algorithms [19] provide dynamic allocation of new fragments on demand. Thus the memory requirement of \( A \)-buffer techniques is virtually unlimited. This is a serious challenge for hardware implementations. As an alternative, multi-pass techniques [139] can be applied, trading off memory requirements for rendering efficiency. We adopt the strategy presented in [56]. Providing only a small fixed number of fragments per pixel, fragments are merged whenever the number of fragments exceeds the preset limit. In our algorithm, each sample that is splatted is processed as follows:

\begin{itemize}
\item \textbf{Accumulate-or-Separate Decision}
\begin{itemize}
\item Using an \( \epsilon \)-\( z \)-threshold as described above, all fragments are checked whether they contain data of the same surface as the new contribution. If this is the case, the contribution is added to the fragment and we are done. Otherwise, the new contribution is treated as a separate surface and a temporary fragment is initialized with its data.
\end{itemize}
\item \textbf{New Fragment Insertion}
\begin{itemize}
\item If the number of fragments including the temporary fragment is smaller than the limit, the temporary fragment is copied into a free slot in the frame buffer and we are done.
\end{itemize}
\item \textbf{Fragment Merging}
\begin{itemize}
\item Otherwise, two fragments have to be merged.
\end{itemize}
\end{itemize}

When fragments are merged, some information is inevitably lost and visual artifacts may occur. These effects are minimized by using an appropriate merging strategy. The main criterion for merging fragments is the difference between their \( z \)-values. This reduces the chance that there are other surfaces, which are going to be rendered later,
that lie between the two merged surfaces. In this case, incorrect back-to-front blending may introduce visual artifacts. On the other hand, if a wrong accumulate-or-separate decision has been taken, multiple fragments for the same surface may have been generated. The above criterion enforces merging of fragments that actually contain data of the same surface. Given the color and alpha values \(c_f, \alpha_f, c_b, \alpha_b\) respectively of the front and back fragment to be merged, the color and alpha values \(c_o, \alpha_o\) of the merged fragment are computed using alpha blending [109]:

\[
\begin{align*}
    c_o &= c_f\alpha_f + c_b\alpha_b(1 - \alpha_f) \\
    \alpha_o &= \alpha_f + \alpha_b(1 - \alpha_f)
\end{align*}
\]

(5.14)

After all points have been splatted, the frame buffer stores several layers of fragments. Hence the normalization pass (see Figure 5.3) is extended by a back-to-front fragment merging pass using Equation 5.14 that generates the final pixel color.

Figure 5.6 shows a geometric object consisting of semi-transparent surfaces, rendered with the multi-layered z-buffer. In most areas, the surfaces are blended flawlessly back-to-front. The geometry of the surfaces, however, is not reconstructed properly around the intersection lines, as illustrated in the close-up in Figure 5.6b. In these regions, contributions of different surfaces are mixed in the fragments, which can cause visual artifacts. Although such problem areas can be reduced by increasing the sampling rate of the surface, they cannot be completely avoided, since the samples do not store any connectivity information.

5.3.3 Edge Antialiasing

In order to perform edge antialiasing (see also Section 2.3), information about partial coverage of surfaces in frame buffer pixels is needed. In partially covered pixels at surface edges, background and foreground surfaces will be blended according to the coverage ratio, leading to a smooth transition. There are various ways of computing this coverage information during rendering. Supersampling methods are popular due to
their simplicity and suitability to hardware implementation, but they come with a seri-
ous overhead in frame buffer memory and bandwidth requirements. These schemes
have been applied to point based rendering algorithms as well [107, 108]. Other meth-
ods attempt to analytically compute coverage instead of sampling it [21], which is
computationally expensive. For point-based representations, a straightforward way to
estimate coverage is to compute the density of points in the frame buffer. The crucial
point in this approach is to come up with a suitable normalization factor that defines
the relation between full coverage and point density [79].

In our approach, coverage information is contained directly in the sum of accumu-
lated resampling filter contributions in each fragment. Consider that for uniform sam-
pling grids, Gaussian filters have an almost perfect flat field response (Section 2.6).
Hence if the filters do not sum up to one at a certain point, this indicates that the point
is not covered completely by reconstruction filters. Even though this does not hold
exactly for nonuniform sampling grids, it is still a useful criterion for determining par-
tial surface coverage. To perform edge antialiasing, we multiply the \( \alpha \)-value of each
fragment by a factor

\[
c = \begin{cases} 
1, & \text{if } \sum \rho_i(x) > 1 \\
\sum \rho_i(x), & \text{otherwise}
\end{cases}, \tag{5.15}
\]

which leads to a gradual blending between background and foreground surfaces in the
fragment merging step. Note that this strategy fails if there is more than one surface
that is partially visible in a pixel. In this case, the visible portions of these surfaces are
not determined correctly. Edge antialiasing is illustrated in Figure 5.7. Figure 5.7a
visualizes the sum of accumulated resampling filter contributions in each pixel, while
Figure 5.7b shows how the foreground surface is blended with the background at sil-
houette edges.

5.4 HARDWARE ACCELERATED EWA SURFACE SPLATTING

In this section, we introduce an alternative formulation of EWA surface splatting,
which we call \textit{object space EWA surface splatting} [112]. Here, we represent the EWA
resampling filter in the local parameter domain rather than in image space as in
Section 5.2. This has the advantage that most of the computations can be implemented
efficiently on modern graphics processing units (GPUs). With the performance of
GPUs increasing faster than the one of CPUs, this algorithm promises interactive ren-
dering of complex objects with millions of samples in the near future.

5.4.1 Resampling Filter in the Local Parameterization Domain

The resampling filter in Equation 5.8 is defined in image space, it corresponds to a
Gaussian \textit{destination domain} resampling filter (see Section 2.7.4). However, the hard-
ware accelerated surface splatting algorithm is based on the resampling filter as a func-
tion in the local parameter domain:

\[
\rho_i(x) = g_{R_i + J_i^T H_j^T (\Pi_i^{-1} (x) - u_i)}, \tag{5.16}
\]
which is called a Gaussian source domain resampling filter (see also Equation 2.53). In contrast to Equation 5.8, this formulation is amenable to hardware acceleration as explained below. Note that the equivalence of Equations 5.8 and 5.16 is derived in Equations 2.54 and 2.55.

5.4.2 Hardware Accelerated Rendering

The hardware accelerated surface splatting algorithm is based on Equation 5.16 and uses a two-pass approach, emulating an $\varepsilon$-z-buffer (Section 5.3). The first pass performs visibility splatting [107] by rendering an opaque polygon for each sample into the z-buffer. The second pass implements Equation 5.16 as follows: The source domain resampling filter is set up as a polygon in object space with a semi-transparent alpha texture, the texture representing a discretized Gaussian function. On modern GPUs, we can implement this using programmable vertex shaders. Then the projection of the textured polygon to image space yields the destination domain resampling filter. The filter is multiplied with the constant surface attributes, i.e., surface color $p_i^c$, and rasterized and accumulated in the framebuffer using graphics hardware.

Representing the EWA Filter as a Textured Polygon. In our hardware accelerated approach, the resampling filter is represented as a textured rectangle in the reference, i.e., fitting, plane $e_i(u_i) = p_i^B(u_i) = p_i + u_i\, 0\, v_{i,0} + u_i\, 1\, v_{i,1}$ of a sample $S_i$. The texture stores a circular unit Gaussian $g_1(s)$, where $s = (s_0, s_1)$ are texture coordinates in the range $[-r...r] \times [-r...r]$. Hence the vertices of the rectangle have texture coordinates $d_j, j = 1...4$:
The rectangle is rotated and scaled around the origin \( u_i = (0,0) \) of the local parameter domain. Hence texture coordinates and coordinates of the local parameterization are related by a linear mapping that consists of a rotation and a scaling

\[
\mathbf{u}_i = \text{Rot}(\theta) \cdot \Lambda \cdot \mathbf{s}, \quad \text{where } \text{Rot}(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \text{ and } \Lambda = \begin{bmatrix} r_0 & 0 \\ 0 & r_1 \end{bmatrix}, \quad (5.18)
\]

and the vertices of the textured rectangle have local coordinates \( \text{Rot}(\theta) \cdot \Lambda \cdot \mathbf{d}_j \), as illustrated in Figure 5.8.

![Figure 5.8](image-url)

**FIGURE 5.8** Mapping the unit Gaussian texture from texture space to the local parameter domain

Since the local parameter domain is mapped to image space by the projection \( \mathbf{x} = \pi_i(\mathbf{u}_i) \) (see Section 5.1), the relation between image space coordinates and texture coordinates is

\[
\mathbf{s} = \Lambda^{-1} \cdot \text{Rot}(\theta)^{-1} \cdot \pi_i^{-1}(\mathbf{x}). \quad (5.19)
\]

This mapping is actually performed by the texture lookups that occur during rasterization of the rectangle in the graphics hardware. According to Equation 2.17, we express the Gaussian texture in image coordinates as:

\[
\frac{1}{r_0 r_1} g_{\pi_i}(\Lambda^{-1} \cdot \text{Rot}(\theta)^{-1} \cdot \pi_i^{-1}(\mathbf{x})) = g_{\text{Rot}(\theta) \cdot \Lambda \cdot \text{Rot}(\theta)}(\pi_i^{-1}(\mathbf{x})), \quad (5.20)
\]

where \( |\Lambda \cdot \text{Rot}(\theta)| = r_0 r_1 \).
To match the resampling filter in the local parameter domain defined by Equation 5.16, we now compute $\text{Rot}(\theta)$ and $\Lambda$ such that

$$g_{\text{Rot}(\theta)}^T \Lambda \cdot \Lambda \cdot \text{Rot}(\theta)(\Pi_i^{-1}(x)) = g_{R_i + J_j H_j^{-1} T}(\Pi_i^{-1}(x)).$$  

(5.21)

This is achieved by determining the eigenvectors and eigenvalues of $R_i + J_j H_j^{-1} T$: The rotation matrix $\text{Rot}(\theta)$ consists of the eigenvectors, and the scaling matrix $\Lambda$ contains the square roots of the eigenvalues. Note that we also have to normalize the texture values by multiplying them with $1/(r_0 r_1)$ as in Equation 5.20.

Finally, the 3D camera space coordinates of the textured rectangle are given by

$$b_j = \left[v_{i,0}^{(0)} v_{i,1}^{(0)}\right] \cdot \text{Rot}(\theta) \cdot \Lambda \cdot d_j, \quad j = 1 \ldots 4.$$  

(5.22)

Visibility Splatting. The purpose of visibility splatting is to render a depth image of the object into the $z$-buffer such that it does not contain any holes [107], as illustrated in Figure 5.9. The depth image will be used to control the accumulation of the semi-transparent splats in the second rendering pass, as described below. For each sample, we render an opaque polygon with vertex coordinates $b_j$ as computed above. The polygon is rasterized into the $z$-buffer only, without modifying other framebuffer attributes, as illustrated in Figure 5.9a. The choice of the polygon vertices $b_j$ ensures that there are no holes in the depth image.

To render a point-based object without artifacts, we must accumulate all the splats of the visible surface closest to the viewer while discarding all other splats. During rasterization of the splats, we decide for each pixel whether to discard or accumulate the current contribution by comparing the depth value of the splat with the depth image that was generated as explained above. However, to prevent contributions of the visible surface from being accidentally discarded, the depth image should be translated away from the viewpoint by a small depth threshold $\varepsilon$. Note that each point in the scene is moved on a viewing ray, as illustrated in Figure 5.9b. This ensures that the occlusion of surface B by surface A is determined correctly. Obviously, visibility splatting may still discard visible surfaces if $\varepsilon$ is too small, or lead to the blending of several surfaces if $\varepsilon$ is too big.

EWA Splatting. To accumulate the sum of resampling filters in the framebuffer, we render the alpha textured rectangles with vertices $b_j$ into the frame buffer using additive alpha blending. By enabling depth comparison with the depth image generated in the first rendering pass while disabling depth update, this has the effect of blending together all splats within the depth range $\varepsilon$.

Finally, we have to normalize the accumulated splats by dividing the values in each framebuffer pixel by the sum of the resampling filter contributions (Equation 5.9). Unfortunately, this operation is not commonly supported in graphics hardware yet. In our implementation, we store the normalization weights in the framebuffer, too. After all splats are rendered, the framebuffer is read back to main memory and normalization is performed in software. As an alternative, we could also apply normalization to each surfel in a pre-process [112]. This allows us to skip per-pixel normalization at the expense of a slight decrease in image quality (see also Section 5.5).
5.5 RESULTS AND COMPARISON

In this section, we first compare the image quality of EWA surface splatting to several other texture filtering methods (Section 5.5.1). In Section 5.5.2, we provide performance data for the software and the hardware implementation.

5.5.1 Image Quality Comparison

We compare the image and antialiasing quality of EWA surface splatting to several forward and backward mapping texture filtering approaches. In forward mapping techniques, each texture sample is processed individually by projecting it to image space and evaluating the reconstruction filter in image space. Hence we also use splatting as an alternative term for forward mapping. In splatting approaches, the number of filter kernels (i.e., splats) that has to be computed is equal to the number of texture samples being rendered. On the other hand, backward mapping techniques process individual pixels and evaluate a reconstruction filter for each pixel. Texture filtering in triangle based rendering pipelines is typically based on backward mapping [143], and it is also used in ray-tracing algorithms [54].

- **Splatting Ellipses:**
  This method is similar to EWA splatting in that it is based on elliptical Gaussian reconstruction filters, which are evaluated and accumulated in image space. However, we do not perform the convolution of the projected reconstruction filters with the low-pass filter. Hence splatting ellipses is equivalent to EWA splatting, except that the texture is not band-limited in image space.

- **Splatting Circles:**
  Here, the resampling filters in screen space are always circular, i.e., their shape is independent of the surface orientation. The radii of the circles are set to match the length of the major radii of corresponding elliptical filters. This guarantees that no holes appear in the reconstruction. Additionally, the radii are clamped to be at least as large as the pixel spacing of the output image to avoid aliasing.

---

FIGURE 5.9 Visibility splatting: (a) Depth image without holes generated by rendering opaque polygons into the z-buffer. (b) Applying the depth offset $\varepsilon$ by translating the scene along the viewing rays.
Bilinear Interpolation:
In contrast to the splatting approaches discussed above, bilinear interpolation is a backward mapping approach. It is a popular and efficient algorithm provided in most hardware rendering pipelines based on triangles. At each pixel, the texture is reconstructed by bilinearly interpolating the values of the four nearest texture samples.

Trilinear Interpolation:
Trilinear interpolation for texture filtering is also known as mip-mapping [138]. In a preprocessing step, a set of so called mip-maps is computed by band-limiting the original texture with low-pass filters with different cutoff frequencies. During rendering, at each pixel two appropriate mip-maps are selected such that high frequencies in the texture exceeding the Nyquist limit of the output sampling grid are avoided. Then, the texture is reconstructed bilinearly in each of the mip-maps. The final pixel value is obtained by linearly interpolating between these samples.

Backward EWA Filtering:
This approach is based on the destination space formulation of the EWA resampling filter, hence it is mathematically equivalent to EWA surface splatting. Introduced by Greene and Heckbert [44] in 1986, it is still used as a benchmark for high quality texture filtering. Since the evaluation of the exact EWA filter is computationally rather expensive, approximations of the exact approach have been proposed for hardware acceleration [84].

While it is straightforward to render nonuniformly sampled, non-planar surfaces with splatting approaches (see Section 5.3), rendering such data using backward mapping requires a ray-tracing approach. Hence, the test data sets that we use in our comparison are uniformly sampled, planar surfaces with a color texture, which can be rendered efficiently using backward mapping. We believe that these datasets are sufficient to compare image and antialiasing quality of the texture filters.

In Figure 5.10, we compare forward and backward EWA filtering with a texture that contains black and white text and that has a resolution of $1634 \times 4056$ samples. The resolution of the output image is $768 \times 480$ pixels. The images generated by forward and backward EWA filtering are visually indistinguishable. Because of the Gaussian low-pass filter integrated in the EWA resampling filter, the images exhibit no apparent aliasing artifacts. This is in contrast to Figure 5.11, where we compare ellipse splatting to bilinear interpolation. Both these approaches do not include a low-pass filter, therefore high frequencies in the texture lead to aliasing. Here, aliasing causes the disintegration of the text texture; in Figure 5.10, a much larger part of the text is readable than in Figure 5.11.

The texture in Figures 5.10 and 5.11 is minified, which means that there is more than one texture sample mapped to each pixel. Under minification, the low-pass behavior of the texture filter dominates the characteristics of the resampling filter. The results of ellipse splatting and bilinear filtering are very similar in this case, since neither approach attempts to band-limit the texture to the Nyquist frequency of the output image. However under magnification, it shows that ellipse splatting with Gaussian kernels leads to a smoother reconstruction of the texture than bilinear filtering. As shown in Figure 5.12, the bilinear filter exhibits staircasing artifacts corresponding to the sampling grid of the input texture. This effect is also called source aliasing (see also Chapter 2).
Figure 5.13 illustrates circle splatting and trilinear mip-mapping. The low-pass behavior of circle splatting is practically identical to EWA splatting, since the EWA resampling filter under minification approximates a circular Gaussian with unit variance (see also Figure 2.14). Remember that in circle splatting a minimum radius of at least the pixel spacing is enforced. This has the effect that the texture in image space is band-limited and aliasing artifacts are avoided. Note that a circular low-pass filter in image space corresponds to an elliptical filter in texture space; i.e., the filter has an elliptical footprint in texture space. Therefore, the texture is filtered anisotropically by circle and EWA splatting. Trilinear interpolation avoids aliasing by accessing pre-filtered textures. However, since the bilinear interpolation step always uses the four near-
5.5 RESULTS AND COMPARISON

FIGURE 5.11 Comparison of ellipse splatting and bilinear filtering: (a) ellipse splatting, (b) bilinear filtering.

FIGURE 5.12 Comparison of elliptical Gaussian and bilinear reconstruction: (a) elliptical Gaussian, (b) bilinear reconstruction.
est texture samples, the texture is filtered *isotropically*. In Figure 5.13b, this leads to excessive blurriness in the horizontal direction in the output image.

We summarize the properties of the texture filters discussed above in Figure 5.14. Here we rendered a zebra texture with a resolution of 768 × 768 texture samples to an output image with 768 × 190 pixels. Figure 5.14a and 5.14d were generated by forward and backward EWA filtering, respectively. Forward and backward mapping leading to visually identical results, EWA filtering avoids aliasing and excessive blurriness by anisotropically filtering the texture. Ellipse splatting and bilinear interpolation (Figure 5.14b and 5.14e) lead to Moiré patterns, since the texture is not band-limited.
Zebra texture rendered with the different texture filters: (a) EWA surface splatting, (b) ellipse splatting, (c) circle splatting, (d) backward EWA filtering, (e) bilinear interpolation, (f) trilinear interpolation.
before sampling. While circle splatting in Figure 5.14c shows similar low-pass behavior as EWA filtering, trilinear interpolation in Figure 5.14d leads to excessive blurriness in the horizontal direction because of isotropic filtering.

As a third example, we have rendered a checkerboard texture with a resolution of $1024 \times 1024$ samples to an output image with $768 \times 500$ pixels in Figure 5.15. Each square in the checkerboard consists of $16 \times 16$ texture samples. While in the above examples only minification occurs, the texture in Figure 5.15 is magnified in the areas closer to the viewpoint. Comparing 5.15a and 5.15b to 5.15f, the excessive blurriness of trilinear filtering in the horizontal direction becomes apparent again. In 5.15c and 5.15d, Moiré patterns appear because of the lack of a low-pass filter. Figure 5.15e reveals the disadvantage of circle splatting: Under magnification, circle splatting leads to much blurrer results than EWA splatting. The EWA splats are dominated by the reconstruction filter under magnification, which are projected to ellipses in screen space. In circle splatting, these are approximated by circles whose radii correspond to the length of the major axis of these ellipses, which results in excessive blurring.

**FIGURE 5.15** Checkerboard textures rendered using different texture filters: (a) EWA surface splatting, (b) backward mapping EWA, (c) ellipse splatting, (d) bilinear filtering, (e) circle splatting, (f) trilinear filtering.
5.5.2 Performance

In this section, we summarize the performance of our software and hardware accelerated implementation of the EWA surface splatting algorithm.

**Software Implementation.** We have implemented a point-sample rendering pipeline based on EWA surface splatting in software. As an object representation, we use the hierarchical LDC tree data structure presented in Chapter 6. However, we disabled hierarchical rendering for the performance tests presented below. In the tests, we used a multi-layered framebuffer with three layers, allowing for edge-antialiasing and rendering of semi-transparent surfaces.

Figure 5.16 depicts our test objects. Figure 5.16a shows a face that was rendered using a point cloud acquired by a laser range scanner. Figures 5.16b and 5.16c show point-sampled geometric objects. We illustrate high quality texturing on terrain data and semi-transparent surfaces on the complex model of a helicopter.

Table 5.1 shows the performance of our unoptimized C implementation of surface splatting. The frame rates were measured on a 1.1 GHz AMD Athlon system with 1.5 GByte memory. The framebuffer had a resolution of $256 \times 256$ and $512 \times 512$ pixels.

**Hardware Implementation.** We have implemented our hardware accelerated point-rendering pipeline with DirectX 8.1. Performance has been measured on a 1 GHz AMD Athlon system with 512 MB memory. We used an ATI Radeon 8500 graphics card and an NVidia GeForce4 Ti 4400 GPU. The first rendering pass requires 13 vertex shader instructions to set up the opaque polygons for visibility splatting, while the computation of the object space EWA filter in the second rendering pass is performed in 78 vertex shader instructions. The hardware accelerated algorithm is based on the LDC tree data structure presented in Chapter 6, too. We also disabled hierarchical rendering when measuring rendering performance.

Figure 5.17 depicts renderings of our test objects, all the point clouds were acquired from geometric data. The rendering performance of the hardware accelerated algorithm is summarized in Table 5.2. We rendered to an output resolution of $256 \times 256$ and $512 \times 512$ pixels and we were able to project approximately three million EWA splats per second to the output resolution of $512 \times 512$ pixels on our test platform. In contrast to the software implementation, the hardware accelerated algorithm is less sensitive to the output resolution.
FIGURE 5.16 Test objects for the software implementation of EWA surface splatting: (a) Matterhorn, (b) terrain data, (c) helicopter. The Matterhorn data set in (a) is courtesy of the Swiss Federal Office of Topography. [see Color Plate 2 on page 158]

TABLE 5.1 Rendering performance of the software implementation of EWA surface splatting for framebuffer resolutions $256 \times 256$ and $512 \times 512$.

<table>
<thead>
<tr>
<th>Data</th>
<th># Points</th>
<th>256 x 256</th>
<th>512 x 512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scanned Head</td>
<td>429075</td>
<td>1.3 fps</td>
<td>0.7 fps</td>
</tr>
<tr>
<td>Matterhorn</td>
<td>4782011</td>
<td>0.2 fps</td>
<td>0.1 fps</td>
</tr>
<tr>
<td>Helicopter</td>
<td>987552</td>
<td>0.6 fps</td>
<td>0.3 fps</td>
</tr>
</tbody>
</table>
5.5 RESULTS AND COMPARISON

FIGURE 5.17  Test objects for the hardware accelerated surface splatting algorithm: (a) chameleon, (b) wasp, (c) salamander, (d) Fiesta. [see Color Plate 3 on page 159]

TABLE 5.2  Rendering performance of the hardware accelerated surface splatting algorithm for frame buffer resolutions $256 \times 256$ and $512 \times 512$. The left performance number was measured with the ATI Radeon 8500 graphics card, the right number corresponds to the NVidia GeForce4 Ti4400 GPU.

<table>
<thead>
<tr>
<th>Data</th>
<th># Points</th>
<th>$256 \times 256$</th>
<th>$512 \times 512$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Salamander</td>
<td>103389</td>
<td>18.9/30.0 fps</td>
<td>16.9/24.9 fps</td>
</tr>
<tr>
<td>Chameleon</td>
<td>101685</td>
<td>17.0/23.2 fps</td>
<td>14.0/19.0 fps</td>
</tr>
<tr>
<td>Wasp</td>
<td>273325</td>
<td>6.1/8.0 fps</td>
<td>5.1/6.1 fps</td>
</tr>
<tr>
<td>Fiesta</td>
<td>352467</td>
<td>5.5/7.0 fps</td>
<td>3.8/5.3 fps</td>
</tr>
</tbody>
</table>
In this chapter, we discuss data structures suitable for representing point-sampled geometry, focusing on rendering as the main operation that is performed on the data. Therefore, appropriate data structures should meet a number of criteria that are important for rendering:

**Storage Efficiency.** To facilitate interactive rendering of large, complex graphics objects, the data structure should make efficient use of limited main memory resources. Therefore, it should have a compact memory footprint, also minimizing data transfer bandwidth from the main memory to the CPU.

**Efficient Warping.** The transformation and projection of the point data to the image plane accounts for a significant amount of processing time in a point rendering pipeline. Data structures can help to minimize this cost by enforcing a certain regularity in the point data that can be exploited to perform efficient incremental calculations during warping.

**Multiresolution Representation.** Depending on the rendered view of a point-sampled object, it is in general not necessary to project all points to the image plane. On the one hand, certain parts of the object, i.e., certain point samples, may not be visible in the desired view. On the other hand, the object may project onto only few output samples, such that a coarser discretization of the object is sufficient for rendering without degrading the image quality. Rendering that is adaptive to the output resolution is also called *level-of-detail* (LOD) rendering. This is achieved by multiresolution data structures that store several discretizations of the object at different resolutions. Hence, a suitable representation can be chosen during rendering depending on the desired view. Further, the data structure should provide spatial access, such that data that is not visible in the rendered view can be skipped quickly.
Complete Object Representation. The data structure should be capable of completely describing the shape and appearance of 3D graphics objects, i.e., it should contain the geometry and reflectance properties of the surfaces. In particular, this means that there should not be any restrictions on the viewpoints that a user can choose for rendering, nor should the image quality depend on the choice of the viewpoint. It should also be possible to change the illumination of the object.

In the following, we restrict ourselves to a discussion of data structures that are tailored for rendering static data. We do not consider other operations, such as neighborhood queries, changing the positions of the sampling points, or dynamically removing and adding samples. Such operations pose different requirements and it is a very challenging task to fulfill all of them in a single data structure. In Section 6.1, we will present the LDC tree data structure, which we have developed for the surfel rendering system [107].

6.1 THE LDC TREE

The LDC tree, which stands for layered depth cube tree, is an image-based data structure designed specifically for efficient point rendering. With the term image-based data structure we denote a data structure that stores a collection of sampled views of an object. A sampled view consists of individual data points without connectivity information, usually on a uniform 2D grid in a 2D image domain. The most popular domain is a planar rectangular area, but cylindrical and spherical domains have been used, too; lately, the concept has also been extended to higher dimensional representations [69, 43].

Image based representations have a long tradition in computer graphics, although their practicability as versatile object descriptions has only been discovered gradually. In contrast to traditional polygonal systems, the main advantage of image based systems is that rendering time is proportional to the number of points in the source images, but independent of scene complexity. Typically, the sampling grids of the source images are adapted to the resolution of the generated views, irrespective of the frequency spectrum of the sampled image signal. Hence, to avoid aliasing problems, proper prefiltering has to be performed before sampling the source images. Often, source images are acquired using physical devices such as digital cameras, which include an inherent filtering process.

Many image based rendering systems have suffered from large memory requirements, noticeable artifacts from certain viewing directions, and an inability to handle dynamic lighting. By extending 2D pixels to 3D surface samples (see also Chapter 4), data structures and algorithms for point rendering feature the computational complexity of image-based rendering with quality, flexibility and memory requirements approaching those of traditional polygon graphics. To preserve the advantages of image-based systems, a similar strategy for sampling 3D geometry is used as discussed above for 2D images: 3D geometry is sampled independent of scene complexity, or surface detail, but adapted to the resolution of generated views. Therefore, 3D geometry should be band-limited properly before sampling to prevent aliasing. Unfortunately, signal processing theory as reviewed in Chapter 2 cannot be extended directly to 2D manifolds. Neither is there a precise definition of frequencies of 2D manifolds, nor a criterion for proper sampling analogous to the sampling theorem described in
Section 2.2. However, these problems are outside the scope of this thesis. Geometric aliasing, i.e., improper sampling and imperfect reconstruction of 3D geometry, may occur in our data structure (see Section 6.1.2).

The rest of this chapter is organized as follows: In Section 6.1.1, we start by briefly reviewing previous work in the area of image-based data structures for rendering, providing the context for our own contribution. Next, we describe the LDC sampling scheme in Section 6.1.2. The hierarchical structure of the LDC tree is covered in Section 6.1.3, and our rendering algorithm is presented in Section 6.1.4.

6.1.1 Previous Work

The most familiar image based data structure in computer graphics are surface textures [22, 48]. Surface textures are images that are mapped onto surfaces of three-dimensional objects. Texture images may contain any surface attribute such as color, reflectance properties [11, 89], transparency [40], or perturbation of the surface normal [12]. However, textures are not a complete object representation per se, because they do not describe the shape of the surfaces.

It has been proposed to use textured planar surfaces to replace complex geometry. The general idea is to render a scene from one or several viewpoints and store the generated views as an intermediate scene representation. Views from nearby camera positions are then approximated by appropriately warping a suitable precomputed image by a 2D affine or projective transformation. These methods, which are known as sprites [130, 66], billboards [75, 122] or impostors [77, 117, 119], exploit spatial and temporal coherence in a sequence of views and they are also useful as LOD techniques, but they do not meet our criteria as complete object representations.

The descriptive power of images is greatly enhanced by adding depth information to each pixel, resulting in images that can be interpreted as height fields. In contrast to planar images, depth images also convey shape information. The depth values usually represent offsets along a direction orthogonal to the base domain, or generalized disparities along viewing rays. With depth images, occlusions with other objects in the scene can be determined correctly and parallax effects can be rendered. However, warping depth images is far more challenging than warping planar images. When depth information is considered, e.g., for perspective mappings, the mapping is not invertible any more. Consequently, holes may appear and visibility problems arise.

McMillan [87] stores depth information as a generalized disparity value per pixel. He derives an equation for perspective mapping that takes the form of a perturbation of the planar warping. He also shows how to resolve visibility correctly by obeying a particular drawing order when warping the image [85, 86]. The same warping technique can also be applied to reproject a cylindrical image to a planar view [88]. Since McMillan’s approach is a forward projecting algorithm that maps points from the source to the destination image, it may leave holes in the output image. Alternative approaches avoid the problem by using a combined forward-backward mapping algorithm [123] or by factoring the perspective mapping into a pre-warp and a planar mapping [99], similar to the shear-warp factorization known from volume rendering [60]. Grossman and Dally [46] propose a data structure for 3D objects that consists of a collection of depth images of the object acquired from different directions. The images are tiled into small blocks, typically eight by eight pixels. Using a greedy algorithm, they add blocks to their representation until the object is sampled sufficiently. Their system
also includes an efficient point-based rendering algorithm based on a hierarchical z-buffer to determine the visibility of rendered points.

Depth images are a single-layer representation. Hence, the representation has to be extended if multiple layers of surfaces are required. To this aim, layered depth images were proposed, which potentially contain multiple depth pixels at each discrete location in the image [123]. Pixels in an LDI are called layered depth pixels, storing a set of depth pixels along one line of sight sorted in front to back order. In the original work [123], the lines of sight share a common center of projection (perspective LDIs). But LDIs can also be constructed from parallel projections, which are then called parallel LDIs.

LDIs facilitate the display not only of parallax, but also of disocclusion effects: when viewing LDIs, surfaces may be exposed that were not visible in the first layer. Rendering LDIs is performed by splatting (see also Chapter 5) individual depth pixels onto the output image plane. For perspective LDIs a z-buffer for resolving visibility is not required, since McMillan’s ordering algorithm [86] can also be used with multiple depth values per pixel.

Originally, LDIs were developed as an image based replacement of complex scene elements that allow efficient rendering from a limited range of viewpoints. However, Max [82] used precomputed multi-layer z-buffers, a concept very similar to LDIs, even earlier for hierarchical rendering of trees. Similar to [82], Lischinski [76] proposed to use a collection of LDIs to completely represent the geometry of 3D objects. His data structure, which he calls the layered depth cube (LDC), consists of three parallel LDIs corresponding to three orthogonal directions. With this configuration, the geometry of 3D objects can be captured completely, allowing the reprojection of the object from any viewpoint. Additionally, Lischinski describes how to perform image based lighting with this structure. Oliveira presented a similar approach to completely sample geometry with a number of LDIs [98]. In contrast to [76], he uses six perspective LDIs with a common center of projection forming a parallelepiped instead of three parallel LDIs. He also extends McMillan’s list ordering algorithm to include multiple perspective LDIs with a common center of projection. Another extension of LDIs is the LDI tree [23]. This representation consists of an octree that stores a single parallel LDI at each node. LDI trees are constructed by resampling an arbitrary number of depth images into a common reference frame. Each image is filtered and inserted into all levels of the octree. Due to its multiresolution structure, the LDI tree facilitates hierarchical rendering, similar to work done before in the context of volume rendering [63].

In their QSplat rendering system [114], Rusinkiewicz and Levoy introduce a bounding spheres hierarchy to represent large point-sampled objects. The hierarchy allows for progressive rendering and streaming. Further, incremental coding of all point attributes optimizes its storage efficiency. Botsch et al. propose a hierarchy of points discretized on voxel grids, arguing that discretizing the point positions does not increase the error of the representation under the assumption that the points are a piece-wise constant approximation of the surface. Incremental coding of the hierarchy reduces the memory requirements to store the position of each point to less than two bits.
6.1.2 LDC Tree Sampling

The LDC tree combines the advantages of the LDC sampling scheme with the hierarchical representation of the LDI tree. We first explain the sampling scheme in this section, followed by the description of the hierarchical tree structure in the next.

LDC tree sampling is based on the same strategy as proposed in [76]: 3D objects are discretized by sampling their surfaces with three parallel LDIs in orthogonal directions, as illustrated in Figure 6.1a. Hence, the image planes of the three LDIs form three sides of a cube. Figure 6.1b depicts the sampling process in 2D, emphasizing that for each intersection of a sampling ray with a surface, front- and back-facing, a sample point is recorded.

Analysis of LDC Sampling. Let us denote the spacing of the uniform quadrilateral sampling grid of the LDIs by \( h_0 \). Further, suppose that the surface being sampled is smooth, where by “smooth” we mean that it has a well defined normal at any point and that the normal does not change quickly. Hence, for the following discussion, we locally approximate the surface with its tangent plane.

From Figure 6.2a it becomes clear that for any surface normal, the angle between the normal and the sampling direction of one of the three orthogonal LDIs is always smaller or equal to \( \alpha_{\text{max}} = \sin((1/3, \sqrt{3}, \sqrt{2}) \). More formally, we denote the angle between a normal \( \mathbf{n} \) and the sampling direction of LDI \( i \) by \( \alpha_i(\mathbf{n}) \), where \( i \in \{1, 2, 3\} \). Then it holds \( \min_i(\alpha_i(\mathbf{n})) \leq \alpha_{\text{max}} \) for any normal direction \( \mathbf{n} \). Assume that the minimum angle is taken for the index \( l \in \{1, 2, 3\} \). Therefore, orthogonally projecting a line segment of unit length with arbitrary orientation from the image plane of LDI \( l \) onto the sampled surface results in a line segment that is not longer than \( 1 / \cos(\alpha_{\text{max}}) = \sqrt{3} \), since

\[
\cos(\alpha_{\text{max}}) = \cos(\sin((1/3, \sqrt{3}, \sqrt{2})) = 1 / \sqrt{3}.
\]

Suppose we project four neighboring sampling points from the image plane of LDI \( l \), which form a square with side length \( h_0 \), onto the sampled surface. Since the scaling
that occurs during the projection is limited to a factor of $\sqrt{3}$, this results in a quadrilateral that can always be contained by a square with sidelines $\sqrt{3}h_0$. This is illustrated in Figure 6.2b, where the angle $\alpha_1(n)$ between the surface normal and LDI 1 obeys the relation $\alpha_1(n) \leq \alpha_{\text{max}}$.

**FIGURE 6.2** Analysis of LDC sampling: (a) For any surface normal, there is always one of the three LDI sampling directions whose angle with the normal is at most $\alpha_{\text{max}}$. (b) Four neighboring sample points on the sampled surface can always be enclosed by a square with sidelines $\sqrt{3}h_0$.

**Nonuniform Reconstruction from LDC Sampling.** Obviously, the sampling pattern produced by LDC sampling on a general surface is nonuniform. But from the above considerations, we conclude that it is reasonable to perform reconstruction from this pattern under the conservative assumption that it is a uniform grid with sample spacing $\sqrt{3}h_0$. In particular, suppose we use radially symmetric reconstruction kernels with a local support of radius $\sqrt{3}h_0$. Then the surface will be covered completely by the reconstruction kernels, or in other words, there will be at least one sample in the support of each reconstruction kernel. This means that there will be no gaps in the reconstruction.

In our implementation, we use Gaussian filters to sample and reconstruct attribute functions on the surface. For LDC sampling, we choose Gaussians $g_{V}$ with a diagonal variance matrix

$$V = \begin{bmatrix} 3h_0 & 0 \\ 0 & 3h_0 \end{bmatrix}. \quad (6.2)$$

During sampling of textured surfaces, we apply the prefiltering approach to nonuniform sampling (see also Section 3.1.3) because of its computational efficiency and antialiasing capabilities.

**Redundant Point Sample Elimination.** In the above analysis of LDC sampling, we were only considering samples of the LDI 1, where the angle between the surface
normal \( \mathbf{n} \) in a locally planar neighborhood and the sampling direction of LDI \( l \) obeys \( \alpha_l(\mathbf{n}) \leq \alpha_{\text{max}} \). I.e., we do not need the sample points of the other two LDIs in this neighborhood to ensure the sampling properties we derived. As a consequence, these samples are redundant and they are not needed for signal reconstruction. To save memory and to optimize rendering efficiency, we remove all sample points \( k \) from the LDI \( i \) which satisfy the following two conditions: First, the angle \( \alpha_l(\mathbf{n}_k) \) between the normal \( \mathbf{n}_k \) of the point \( k \) and the sampling direction of LDI \( i \) is larger than \( \alpha_{\text{max}} \); second, the angles \( \{\alpha_j(\mathbf{n}_j) \mid j \in N_k\} \) between the normals of the points \( j \) belonging to the eight-neighborhood \( N_k \) (Figure 6.3a) of the point \( k \) in the LDI \( i \) are all larger than \( \alpha_{\text{max}} \), too. This heuristic leads to a seam of overlapping samples at the boundary between samples of two LDIs, avoiding holes due to point sample removal in these regions. However, the heuristics assumes that the size of surface detail is at least twice the LDI sampling spacing \( h_0 \). Otherwise, it may fail to sample surfaces correctly, as illustrated with a 2D example in Figure 6.3b. We call this problem geometric aliasing; it could be addressed by some kind of geometric filtering, but this is beyond the scope of this thesis.

We illustrate the effect of the redundant sample elimination strategy in Figure 6.4. Points from the three LDIs are colored differently. The object on the left was sampled without redundant sample elimination, it contains 134'528 points. On the right, the redundant sampling elimination technique was applied, reducing the number of points to 98'732, which is 73.4% of the original data.

6.1.3 Octree Structure

To build up a hierarchical octree data structure, we decompose the set of samples \( S \) acquired by LDC sampling into a sequence of nested subsets \( S_{N-1} \subset S_{N-2} \subset \cdots \subset S_0 = S \). The subsets \( S_k \) are constructed by uniformly subsampling each of the three original LDIs by a factor of \( 2^k \), as illustrated in Figure 6.5. Each level \( 0 \leq k < N \) of the LDC tree consists of an LDC that contains the samples of the three subsampled LDIs on level \( k \). Furthermore, the LDCs on each level are partitioned into cubic blocks with an arbitrary resolution of \( b \times b \) sampling positions on the subsampled LDIs. This is shown in Figure 6.6 with a 2D example. Note that certain blocks do
not contain any sample points, therefore they are not stored in the data structure. Hence, each block is an LDC consisting of three orthogonal LDIs with resolution $b \times b$. Also, the reconstruction kernels associated with the points on level $k$ are scaled accordingly, i.e., their radii are $\sqrt[3]{h_0} \cdot 2^k$.

Finally, the non-empty blocks on all levels are connected to a tree structure, as illustrated in Figure 6.7 with the 2D example. In 3D, each parent block on level $k$ has eight child blocks on level $k + 1$.

The number of points that is stored in each layered depth pixel (i.e., the number of surface intersections) is different for each pixel; the number is not bounded, but there also may be no intersection point recorded at a pixel at all. Therefore, we apply an extended
6.1 THE LDC TREE

A sparse matrix data structure to save memory and to allow efficient access to the points during rendering. As illustrated in Figure 6.8, for each LDI of a block the point data is stored in a linear array (data), which is accessed using three auxiliary indexing arrays. The number of entries in the data array corresponds to the total number of sample points in the LDI. The rowIndex array contains the indices of the last point for each row in the block (therefore, this array has a constant length of \( b \) entries). The colIndex array stores the column indices of the LDI pixels, and the nPoints array records the number of points belonging to each LDI pixel. Hence, the length of these arrays corresponds to the number of non-empty pixels in the LDI. Pseudocode for accessing all points in the LDI and reconstructing their pixel coordinates is given in Figure 6.9.

**FIGURE 6.6** On each level, the LDC tree is partitioned into blocks with resolution \( b \times b \), indicated using different shades of gray (2D example). Empty blocks are white.

**FIGURE 6.7** The blocks are connected to a quadtree structure (2D example).
6.1.4 Hierarchical Rendering

The LDC tree supports multiresolution, level-of-detail, and progressive rendering with hierarchical view frustum and hierarchical backface culling. The rendering algorithm is summarized in pseudocode in Figure 6.10. We perform a top-down traversal of the LDC tree, starting at the lowest resolution at its root. For each block that is visited, we decide whether it can be clipped because it lies completely outside the view frustum, or whether it can be backface culled. In both cases, the block does not have to be processed further and the subtree attached to the block can be skipped entirely. Otherwise, we check per block whether the projected reconstruction kernels (that are associated with the points in the block) in screen space match a rendering quality criterion. Typically, the projected reconstruction kernels should not be larger than the size of a pixel. In other words, there should be at least one reconstruction kernel (i.e., sample point) per pixel. If the quality criterion is not met and the block is not a leaf node in the tree, we continue by traversing the child nodes of the block. Otherwise, we render the cur-

**FIGURE 6.8** Illustration of the extended sparse matrix data structure for storing LDIs: (a) LDI with resolution $4 \times 4$ (i.e., $b = 4$), numbers in the LDI pixels indicate the number of points per pixel; (b) corresponding extended sparse matrix structure.

point = 0;
pixel = 0;
row = 0;

while (row < b) {
    while (point < rowIndex[row]) {
        col = colIndex[pixel];
        n = nPoints[pixel];
        while (n) {
            // data[point] contains the current point data
            // at the LDI pixel [row, col]
            point++;
            n--;
        }
        pixel++;
    }
    row++;
}

**FIGURE 6.9** Pseudocode for accessing the sample points of an LDI stored in the extended sparse matrix data structure.

6.1.4 Hierarchical Rendering

The LDC tree supports multiresolution, level-of-detail, and progressive rendering with hierarchical view frustum and hierarchical backface culling. The rendering algorithm is summarized in pseudocode in Figure 6.10. We perform a top-down traversal of the LDC tree, starting at the lowest resolution at its root. For each block that is visited, we decide whether it can be clipped because it lies completely outside the view frustum, or whether it can be backface culled. In both cases, the block does not have to be processed further and the subtree attached to the block can be skipped entirely. Otherwise, we check per block whether the projected reconstruction kernels (that are associated with the points in the block) in screen space match a rendering quality criterion. Typically, the projected reconstruction kernels should not be larger than the size of a pixel. In other words, there should be at least one reconstruction kernel (i.e., sample point) per pixel. If the quality criterion is not met and the block is not a leaf node in the tree, we continue by traversing the child nodes of the block. Otherwise, we render the cur-
rent block by projecting the points to screen space and using the splatting method described in Chapter 5 to perform image reconstruction.

\[
cur = root;
while( !lastNode(cur) ) {
    if( viewFrustumCulling(cur) OR backFaceCulling(cur) ) {
        skipSubtree(cur);
    } else {
        if( qualitySatisfied(cur) OR isLeaf(cur) ) {
            render(cur);
            skipSubtree(cur);
        } else {
            // step to the next block in depth-first order
            getNext(cur);
        }
    }
}
\]

**FIGURE 6.10** Pseudocode for hierarchical rendering using the LDC tree.

**View Frustum Culling.** To perform per-block view frustum culling, we use axis aligned bounding boxes for the blocks. The \( x \), \( y \), and \( z \) coordinates of the bounding boxes of blocks on level \( k \) are given by the minimum and maximum \( x \), \( y \), and \( z \) coordinates respectively of the points contained by the block with an additional offset of \( \sqrt{3} h_0 \cdot 2^k \) along all coordinate axes, which corresponds to the radius of the filters on level \( k \) (Section 6.1.2).

We transform the perspective viewing frustum into a normalized viewing frustum, similar as, e.g., in OpenGL [143], to simplify the intersection calculations. Intersection tests of individual blocks with the normalized view frustum are computed using separation axis testing, similar as described in [31]. This algorithm is based on the fact that two convex polyhedra do not intersect if there exists a line such that the projections of the polyhedra onto the line do not intersect. It is easy to see that in this case there exists a plane with a normal direction corresponding to the direction of the line that separates the two polyhedra. However, the more difficult problem is selecting a finite set of line directions that are sufficient to determine intersection or non-intersection. For convex polyhedra it turns out that the set consisting of face normals for the two polyhedra and vectors that are the cross product of edges, one edge from each polyhedron, is sufficient. In our application, the set contains 26 directions that have to be tested. In an optimized implementation, many intermediate results of the tests can be cached and reused for several directions.

If the bounding block of a box does not intersect the view frustum, the block is culled and the subtree attached to it is not further traversed during rendering. However, we do not clip blocks to the view frustum. Blocks that partially lie within the view frustum are sent further down the rendering pipeline as a whole; clipping is performed on a per-point basis.

**Backface Culling.** Backface culling is based on the observation that, similar to polygon rendering, there is no need to render points whose normals are pointing away from the viewer. In other words, the tangent plane of each point defines a half space from
which the point is not visible. Given a set of points, the intersection of their respective half spaces defines a region from which none of the points is visible. We could decide whether the set of points is visible from a given viewpoint by testing the half spaces of all points individually. Instead, to simplify calculations, we approximate the region defined by the intersection of the half spaces by a cone inside it. This visibility cone, which has also been used by [46], allows us to efficiently perform a conservative visibility test for all points of a block.

Let us define the axis of the cone parametrically by the direction \( \mathbf{n} \) of the axis, a point \( \mathbf{p} \) on the axis, and the parameter \( t \):

\[
\mathbf{p} + t \mathbf{n}, \quad \text{where } t \in \mathbb{R}.
\]

We choose \( \mathbf{p} \) heuristically as the average of the points \( \mathbf{p}_i \in P \), where \( P \) is the set of points contained in a block:

\[
\mathbf{p} = \frac{\sum \mathbf{p}_i}{|P|}, \quad \text{where } \mathbf{p}_i \in P.
\]

To determine the direction \( \mathbf{n} \) and the apex angle \( \alpha \) of the visibility cone, we use a different strategy than [46]. We first compute a normal cone containing all (unit) normals \( \mathbf{n}_i \) belonging to the \( \mathbf{p}_i \) (Figure 6.11): We attach the normals at the origin and determine the smallest sphere that encloses all endpoints of the normals (which lie on the unit sphere). The relation between the radius \( r \) of the sphere and the apex angle \( \beta \) of the normal cone is \( r = \sin((1/2)\beta) \), where \( 0 \leq r \leq 1 \) by construction; the axis \( \bar{\mathbf{n}} \) of the normal cone is given by the line from the origin through the center of the sphere.

![Figure 6.11](image.png)  

**Figure 6.11** Construction of the normal cone.

The axis direction and apex angle of the visibility cone are now given by \( \mathbf{n} = -\bar{\mathbf{n}} \) and \( \alpha = \pi - \beta \). As can be seen from Figure 6.12, this guarantees that the visibility cone attached to the origin completely lies within the intersection of all backfacing half spaces of the normals \( \mathbf{n}_i \). On the other hand, no cone with an apex angle larger than \( \pi - \beta \) fits in this region.

Finally, we find the apex \( \mathbf{q} \) of the visibility cone. Since the apex must lie on the line defined by Equation 6.3, it is of the form \( \mathbf{q} = \mathbf{p} + t_0 \mathbf{n} \). Further, \( \mathbf{q} \) must lie in the back-facing half space of all normals, hence \( t_0 \) must satisfy
for each $i$. Therefore we set

$$t_0 = \min_i \frac{(\mathbf{p} - \mathbf{p}_i) \cdot \mathbf{n}_i}{\mathbf{n} \cdot \mathbf{n}_i}.$$  

(6.6)

So to test whether a block can be culled, we check if the viewpoint $\mathbf{x}$ lies within the visibility cone, i.e.:

$$\frac{(\mathbf{x} - \mathbf{q}) \cdot \mathbf{n}}{|\mathbf{x} - \mathbf{q}|} \leq \cos \alpha.$$  

(6.7)

**Evaluation of Quality Criterion.** Since surface point samples discretize all surface properties, i.e., geometry (position, normal) as well as appearance (color, reflectance properties), at the same rate, the quality of images generated by point based rendering largely depends on the projected sample density in image space. The smaller the number of samples that is projected to each output pixel, the less detailed is the reconstruction of the surface geometry and texture. For high quality images, the projected sampling rate of the surface should be at least as high as the sampling rate of the output image, otherwise rendered images look blurry. In other words, at least one point should be projected to each pixel, or equivalently, the radius of the corresponding projected reconstruction kernels should be smaller than the pixel spacing.

In our approach, we conservatively estimate the projected size of the reconstruction kernels per block. We then decide per block, whether the LDC tree should be traversed further to the next higher resolution, or whether it is sufficient to render the current resolution. To get an upper bound on the projected size of the reconstruction kernels of a block on level $k$, we first find the vertex of the block bounding box that is closest to the viewer. Assume that this vertex has a positive $z$-coordinate $z_{min}$. Then the size of the projected reconstruction kernels of the block is bounded by...
where \( s \) accounts for scaling that occurs in the modeling, viewing, and viewport transformations. Note that if these transformations include nonuniform scaling, we conservatively approximate them with uniform scaling factors. We traverse the LDC tree until \( r_p \) is smaller than a user defined quality constant \( c_Q \), or until we reach a leaf. Good image quality is obtained by choosing \( c_Q = 1 \); a smaller value leads to better quality, but also to higher rendering cost.

In Figure 6.13, we illustrate level-of-detail (LOD) rendering using our approach. Figure 6.13a shows a textured salamander model that is represented by an LDC tree. It contains 83,690 sample points on the highest resolution (level \( k = 0 \)), the blocks have a resolution of \( b = 16 \). The rendered image covers a rectangle of 194 × 148 pixels. In Figure 6.13b we illustrate the level in the LDC tree that is selected for rendering: In the foreground the highest resolution level \( k = 0 \) (light gray) is selected, in the background level \( k = 1 \) (dark gray). On level zero, 188 blocks containing 33,340 points were projected, on level one there were 52 blocks and 11,378 points. We visualize the radii of the projected reconstruction kernels using a gray scale mapping in Figure 6.13c. Moving from the tail of the salamander towards its nose, reconstruction kernel radii increase due to decreasing distance from the viewer. As the radii reach an upper limit (here we used \( c_Q = 1 \)), the next higher resolution level in the LDC tree is used, resulting in kernels with \( r_p \approx c_Q/2 \) approximately. Hence we reduce the number of projected samples, i.e., the rendering time, while maintaining the desired image quality.

We can also use the hierarchy of the LDC tree to control the rendering cost. In a typical application, the user specifies a frame rate which is maintained by automatically adapting the image quality parameter \( c_Q \) during user interaction. The image is refined progressively by decreasing \( c_Q \) as soon as the user has chosen the desired viewpoint. This is illustrated in Figure 6.14, where we show a chameleon model that was rendered with three different quality parameters, \( c_Q = 4 \), \( c_Q = 2 \), and \( c_Q = 1 \). Note that to further increase rendering efficiency, we also decreased the resolution of the output image according to \( c_Q \).

The resulting frame rates and number of points that are projected in each frame are summarized in Table 6.1. The performance numbers were measured on a PC class computer with a 2 GHz Pentium IV processor and 1 GByte of main memory. We used the surface splatting rendering algorithm described in Chapter 5, the resolution of the output image was 384 × 384 pixels.

**Incremental Block Warping.** A fundamental operation that is performed in any point rendering pipeline (see also Chapter 7 and 5) is the projection of the 3D points to the 2D output image plane. This usually involves affine 3D-to-3D transformations, such as the modeling and the viewing transform, a 3D-to-2D perspective projection to the image plane, and the 2D-to-2D viewport transformation, which maps the projected points to output image coordinates. This is similar to conventional rendering pipelines such as OpenGL [143].

For a general 3D point, these operations can be computed using a multiplication of a matrix \( 3 \times 3 \) by a \( 3 \times 1 \) vector for the affine 3D transformation, followed by two divisions for the perspective projection, and two additions and four multiplications for
the viewport transformation. However in an LDI, the $x$- and $y$-coordinates of all points lie on a uniform grid. We can exploit this regularity of LDI sampling to simplify the calculations for points stored in the LDC data structure. The core idea is to incrementally project points that lie on each LDI. We also apply similar optimizations as proposed by Grossman [46]. Hence, the cost of projecting each point to the output image is reduced to 6 additions, 3 multiplications, and 1 reciprocal.

**FIGURE 6.13** LOD Rendering: (a) Salamander model with color texture. (b) Visualization of the block levels of the LDC tree that are projected. In the foreground, level $k = 0$ is used, and in the background level $k = 1$. (c) Visualization of the projected reconstruction filter radii.

**FIGURE 6.14** Rendering with different quality parameters: (a) $c_Q = 4$, (b) $c_Q = 2$, (c) $c_Q = 1$. 
### TABLE 6.1
Rendering of the chameleon model with different quality parameters.

<table>
<thead>
<tr>
<th></th>
<th>$c_Q=4$</th>
<th>$c_Q=2$</th>
<th>$c_Q=1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frames per second</td>
<td>66.1</td>
<td>18.5</td>
<td>4.6</td>
</tr>
<tr>
<td>Points on level $k = 0$</td>
<td>0</td>
<td>3033</td>
<td>98732</td>
</tr>
<tr>
<td>Points on level $k = 1$</td>
<td>739</td>
<td>26095</td>
<td>0</td>
</tr>
<tr>
<td>Points on level $k = 2$</td>
<td>7108</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total number of points</td>
<td>7847</td>
<td>29128</td>
<td>98732</td>
</tr>
</tbody>
</table>
EWA Volume Splatting

Volume rendering is an important technique in visualizing acquired and simulated datasets in scientific and engineering applications. The ideal volume rendering algorithm reconstructs a continuous function in 3D, transforms this 3D function into screen space, and then evaluates opacity integrals along lines-of-sight. In 1989, Westover [135, 136] introduced splatting for interactive volume rendering, which approximates this procedure. Splatting algorithms interpret volume data as a set of particles that are absorbing and emitting light. Line integrals are precomputed across each particle separately, resulting in footprint functions. Each footprint, or splat, spreads its contribution in the image plane. These contributions are composited back to front into the final image.

In this chapter, we present a technique for high quality volume splatting (see also [147, 148]). Our approach is based on Gaussian resampling filters similar as in Chapter 5, therefore we call the algorithm EWA volume splatting. The main feature of EWA volume splatting is that it integrates an elliptical Gaussian reconstruction kernel and a low-pass filter, therefore preventing aliasing artifacts in the output image while avoiding excessive blurring.

EWA volume splatting works with arbitrary elliptical Gaussian reconstruction kernels and efficiently supports perspective projection. Our method is based on a novel approach to compute the footprint function, which relies on the transformation of the volume data to so-called ray space. This transformation is equivalent to perspective projection. By using its local affine approximation at each voxel, we derive an analytic expression for the EWA footprint in image space. The EWA volume splat primitive can be integrated easily into conventional volume splatting algorithms. Because of its flexibility, it can be utilized to render rectilinear, curvilinear, or unstructured volume datasets. The rasterization of the footprint is performed using forward differencing, requiring only one 1D footprint table for all reconstruction kernels and any viewing direction.
7.1 VOLUME RESAMPLING

We distinguish two fundamental approaches to volume rendering: backward mapping algorithms that shoot rays through pixels on the image plane into the volume data, and forward mapping algorithms that map the data onto the image plane. In the following discussion, we will describe a forward mapping technique. Mapping the data onto the image plane involves a sequence of intermediate steps where the data is transformed to different coordinate systems, as in conventional rendering pipelines. We introduce our terminology in Figure 7.1. Note that the terms space and coordinate system are synonymous. The figure summarizes a forward mapping volume rendering pipeline, where the data flows from the left to the right.

As an overview, we briefly describe the coordinate systems and transformations that are relevant for our technique. We will deal in detail with the effect of the transformations in Section 7.1. The volume data is initially given in source space, which is usually called object space in this context. To render the data from an arbitrary viewpoint, it is first mapped to camera space using the viewing transformation. The camera coordinate system is defined such that its origin is at the center of projection.

We further transform the data to ray space, which is introduced in Section 7.1.1. Ray space is a non-cartesian coordinate system that enables an easy formulation of the volume rendering equation. In ray space, the viewing rays are parallel to a coordinate axis, facilitating analytical integration of the volume function. We present the transformation from camera to ray space in Section 7.1.3; it is a key element of our technique. Since its purpose is similar to the projective transform used in rendering pipelines such as OpenGL, it is also called the projective mapping.

Evaluating the volume rendering equation results in a 2D image in image space. In a final step, this image is transformed to viewport coordinates. Focusing on the essential aspects of our technique, we are not covering the viewport transformation in the following explanations. However, it can be easily incorporated in an implementation. Moreover, we do not discuss volume classification and shading in a forward mapping pipeline, but refer to [94] or [141] for a thorough discussion.

7.1.1 Splatting Algorithms

We review the low albedo approximation of the volume rendering equation [57, 81] as used for fast, direct volume rendering [136, 61, 94, 68]. Figure 7.2a illustrates the corresponding situation in 2D. Starting from this form of the rendering equation, we dis-
cuss several simplifying assumptions leading to the well known *splatting* formulation. Because of their efficiency, splatting algorithms \[136, 94\] belong to the most popular forward mapping volume rendering techniques.

We slightly modify the conventional notation, introducing our concept of ray space. We denote a point in ray space by a column vector of three coordinates $x = (x_0, x_1, x_2)^T$. Given a center of projection and a projection plane, these three coordinates are interpreted geometrically as follows: The coordinates $x_0$ and $x_1$ specify a point on the projection plane. The ray intersecting the center of projection and the point $(x_0, x_1)$ on the projection plane is called a viewing ray.

Using the abbreviation $x = (x_0, x_1)^T$, we also refer to the viewing ray passing through $(x_0, x_1)$ as $x$. The third coordinate $x_2$ specifies the Euclidean distance from the center of projection to a point on the viewing ray. Note that our notation does not distinguish between a ray $x$ and a point in ray space $x$, however it will be clear from the context which one is meant. Furthermore, to simplify the notation, we will use any of the synonyms $x$, $(x, x_2)^T$, or $(x_0, x_1, x_2)^T$ to denote a point in ray space.

The volume rendering equation describes the light intensity $I_\lambda(x)$ at wavelength $\lambda$ that reaches the center of projection along the ray $x$ with length $L$:

$$ I_\lambda(x) = \int_0^L c_\lambda(x, e) f'_c(x, e) e^{-\int_0^\epsilon (f'_c(x, \mu)d\mu)} d\epsilon $$

\[7.1\]
where \( f_c'(x) \) is the extinction function that defines the rate of light occlusion, and \( c_\lambda(x) \) is an emission coefficient. The exponential term can be interpreted as an attenuation factor. Finally, the product \( c_\lambda(x)f_c'(x) \) is also called the source term [81], describing the light intensity scattered in the direction of the ray \( x \) at the point \( x_i \). In the following equations we will omit the parameter \( \lambda \), implying that Equation 7.1 has to be evaluated for different wavelengths separately.

Now we assume that the extinction function in object space (i.e., source space) \( f_c(u) \) is given in the form

\[
f_c(u) = \sum_k w_k r_k(u)
\]

(7.2)

as a weighted sum of coefficients \( w_k \) and reconstruction kernels \( r_k(u) \) (similar as Equation 3.5). This corresponds to a physical model where the volume consists of individual particles that absorb and emit light. The reconstruction kernels \( r_k \) reflect position and shape of individual particles. The particles can be irregularly spaced and may differ in shape, hence the model is not restricted to regular datasets. Note that the extinction function in ray space \( f_c'(x) \) is computed by concatenating a mapping \( \phi \) from object space to camera space and a mapping \( \varphi \) from camera space to ray space (see Figure 7.1), yielding:

\[
f_c'(x) = f_c(\phi^{-1}(\varphi^{-1}(x))) = \sum_k w_k r_k'(x),
\]

(7.3)

where \( r_k'(x) = r_k(\varphi^{-1}(\phi^{-1}(x))) \) is a reconstruction kernel in ray space. The mappings \( \phi \) and \( \varphi \) will be discussed in detail in Section 7.1.3.

Because of the linearity of integration, substituting Equation 7.3 into Equation 7.1 yields

\[
I(x) = \sum_k \left\{ \int_0^L \int \left[ e^{\int r_j'(x, \mu) d\mu} \prod_j e^{\int r_j'(x, \varepsilon) r_j'(x, \varepsilon) \varepsilon} \right] d\varepsilon \right\},
\]

(7.4)

which can be interpreted as a weighted sum of projected reconstruction kernels.

To compute \( I(x) \) numerically, splatting algorithms make several simplifying assumptions, illustrated in Figure 7.2b. Usually the reconstruction kernels \( r_k'(x) \) have local support. The splatting approach assumes that these local support areas do not overlap along a ray \( x \), and the reconstruction kernels are ordered front to back. We also assume that the emission coefficient is constant in the support of each reconstruction kernel along a ray, hence we use the notation \( c_k(x_0, x_1) = c(x_0, x_1, x_2) \), where \( (x_0, x_1, x_2) \) is in the support of \( r_k' \). Moreover, we approximate the exponential function with the first two terms of its Taylor expansion, thus \( e^{-x} \approx 1 - x \). Finally, we ignore self-occlusion. Exploiting these assumptions, we rewrite Equation 7.4, yielding:
where $q_k(x)$ denotes an integrated reconstruction kernel, hence:

$$
q_k(x) = \int r_k'(x, x_2)dx_2.
$$

Equation 7.5 is the basis for all splatting algorithms. Westover [136] introduced the term footprint function for the integrated reconstruction kernels $q_k$. The footprint function is a 2D function that specifies the contribution of a 3D kernel to each point on the image plane. Since integrating a volume along a viewing ray is analogous to projecting a point on a surface onto the image plane, the coordinates $x = (x_0, x_1)^T$ are also called image coordinates, and we say that $I(x)$ and $q_k(x)$ are defined in image space.

Splatting is attractive because of its efficiency, which it derives from the use of pre-integrated reconstruction kernels. Therefore, during volume integration, each sample point along a viewing ray is computed using a 2D convolution. In contrast, ray-casting methods require a 3D convolution for each sample point. This provides splatting algorithms with an inherent advantage in rendering efficiency. Moreover, splatting facilitates the use of higher quality kernels with a larger extent than the trilinear kernels typically employed by ray-casting. On the other hand, basic splatting methods are plagued by artifacts because of incorrect visibility determination. This problem is unavoidably introduced by the assumption that the reconstruction kernels do not overlap and are ordered back to front. It has been successfully addressed by several authors [136, 93]. In contrast, our main contribution is a novel splat primitive that provides high quality antialiasing and efficiently supports elliptical kernels. We believe that our novel primitive is compatible with all state-of-the-art splatting algorithms.

### 7.1.2 The Volume Resampling Filter

The splatting equation (Equation 7.5) represents the output image as a continuous image space signal $I(x)$. In order to properly sample this function to a discrete output image without aliasing artifacts, it has to be band-limited to match the Nyquist frequency of the discrete image. According to signal processing theory (Chapter 2), we achieve this band-limitation by convolving $I(x)$ with an appropriate low-pass filter $h(x)$, yielding the antialiased splatting equation

$$
I(x) = (I \otimes h)(x) = \sum_k w_k \int c_k(\eta)q_k(\eta) \prod_{j=0}^{k-1} (1 - w_j q_j(\eta))h(x - \eta)d\eta.
$$

Unfortunately, the convolution integral in Equation 7.7 cannot be computed explicitly because of the emission and attenuation terms. Hence we make two simplifying assumptions to rearrange it, leading to an approximation that can be evaluated efficiently.

First, we assume that the emission coefficient is approximately constant in the support of each footprint function $q_k$, hence $c_k(x) \approx c_k$ for all $x$ in the support area. Together with the assumption that the emission coefficient is constant in the support of
each reconstruction kernel along a viewing ray, this means that the emission coefficient is constant in the complete 3D support of each reconstruction kernel. In other words, this corresponds to per-voxel evaluation of the shading model, or pre-shading [94], ignoring the effect of shading for antialiasing. Note that prefiltering methods for surface textures usually ignore aliasing due to shading, too.

Additionally, we assume that the attenuation factor has an approximately constant value \( o_k \) in the support of each footprint function. Hence:

\[
\prod_{j=0}^{k-1} (1 - w_j q_j(x)) \approx o_k
\]  

for all \( x \) in the support area. A variation of the attenuation factor indicates that the footprint function is partially covered by a more opaque region in the volume data. Therefore this variation can be interpreted as a “soft” edge. Ignoring such situations means that we cannot prevent edge aliasing. Again, this is similar to rendering surfaces, where edge and texture aliasing are handled by different algorithms as well.

Exploiting these simplifications, we can rewrite Equation 7.7 to:

\[
(I \otimes h)(x) \approx \sum_k w_k c_k o_k \int_{\mathbb{R}^2} q_k(\eta) h(x - \eta) d\eta
\]

\[
= \sum_k w_k c_k o_k (q_k \otimes h)(x)
\]  

Following the terminology of Section 2.7 we call

\[
\rho_k(x) = c_k o_k (q_k \otimes h)(x) = (p_k \otimes h)(x)
\]

an ideal volume resampling filter, combining a projected reconstruction kernel \( p_k = c_k o_k q_k \) and a low-pass kernel \( h \). Hence, we can approximate the antialiased splatting equation (Equation 7.7) by replacing the footprint function \( q_k \) in the original splatting equation (Equation 7.5) with the resampling filter \( \rho_k \). This means that instead of band-limiting the output function \( I(x) \) directly, we band-limit each footprint function separately. Under the assumptions described above, we get a splatting algorithm that produces a band-limited output function respecting the Nyquist frequency of the raster image, therefore avoiding aliasing artifacts. Remember that the reconstruction kernels are integrated in ray space, resulting in footprint functions that vary significantly in size and shape across the volume. Hence the resampling filter in Equation 7.10 is strongly space variant.

Swan et al. presented an antialiasing technique for splatting [127] that is based on a uniform scaling of the reconstruction kernels to band-limit the extinction function. Their technique produces similar results as our method for radially symmetric kernels. However, for more general kernels, e.g., elliptical kernels, uniform scaling is a poor approximation of ideal low-pass filtering. Aliasing artifacts cannot be avoided without introducing additional blurriness. On the other hand, our method provides non-uniform scaling in these cases, leading to superior image quality as illustrated in Section 7.3. Moreover, our analysis above shows that band-limiting the extinction function does
not guarantee alias free images. Because of shading and edges, frequencies above the Nyquist limit persist. However, these effects are not discussed in [127].

### 7.1.3 The EWA Volume Resampling Filter

In this section, we first describe how to map arbitrary elliptical Gaussian volume reconstruction kernels from object to ray space. Our derivation results in an analytic expression for the kernels in ray space \( r_k(x) \) as in Equation 7.3. We will then be able to analytically integrate the kernels according to Equation 2.20 (Section 2.6) and to convolve the footprint function \( q_k \) with a Gaussian low-pass filter \( h \) as in Equation 2.18, yielding an elliptical Gaussian resampling filter \( \rho_k \).

#### The Viewing Transformation.

The reconstruction kernels are initially given in source space, or object space, which has coordinates \( u = (u_0, u_1, u_2)^T \). As in Section 7.1.1 we denote the Gaussian reconstruction kernels in object space by:

\[
r_k(u) = g_{R_k}(u - u_k), \tag{7.11}
\]

where \( u_k \) are the voxel positions in object space. For regular volume datasets, the variance matrices \( R_k \) are usually identity matrices. For rectilinear datasets, they are diagonal matrices where the matrix elements contain the squared distances between voxels along each coordinate axis. Curvilinear and irregular grids have to be resampled to a more regular structure in general. For example, Mao et al. [78] describe a stochastic sampling approach with a method to compute the variance matrices for curvilinear volumes.

We denote camera coordinates by a vector \( t = (t_0, t_1, t_2)^T \). Object coordinates are transformed to camera coordinates using a mapping \( t = \varphi(u) \), called viewing transformation. The viewing transformation is usually an affine mapping defined by a matrix \( W \) and a translation vector \( d \) as \( \varphi(u) = Wu + d \).

#### The Projective Transformation.

We will concatenate the viewing transformation with a projective transformation that converts camera coordinates to ray coordinates as illustrated in Figure 7.3. Camera space is defined such that the origin of the camera coordinate system is at the center of projection and the projection plane is the plane \( t_2 = 1 \). Camera space and ray space are related by the mapping \( x = \phi(t) \). Using the definition of ray space from Section 7.1.1, \( \phi(t) \) and its inverse \( \phi^{-1}(x) \) are therefore given by:

\[
\begin{bmatrix}
x_0 \\
x_1 \\
x_2
\end{bmatrix} = \phi(t) = \begin{bmatrix}
t_0/t_2 \\
t_1/t_2 \\
(t_0, t_1, t_2)^T
\end{bmatrix}, \tag{7.12}
\]

\[
\begin{bmatrix}
t_0 \\
t_1 \\
t_2
\end{bmatrix} = \phi^{-1}(x) = \begin{bmatrix}
x_0/1 \cdot x_2 \\
x_1/1 \cdot x_2 \\
1/1 \cdot x_2
\end{bmatrix}
\]

where \( l = \|(x_0, x_1, 1)^T\| \).
Unfortunately, these mappings are not affine, so we cannot apply Equation 2.17 (Section 2.6) directly to transform the Gaussian reconstruction kernels from camera to ray space. To solve this problem, we introduce the local affine approximation $\phi_k$ of the projective transformation. It is defined by the first two terms of the Taylor expansion of $\phi$ at the point $t_k$:

$$\phi_k(t) = x_k + J_k \cdot (t - t_k),$$

(7.13)

where $t_k$ is the center of a Gaussian in camera space and $x_k = \phi(t_k)$ is the corresponding position in ray space. The Jacobian $J_k$ is given by the partial derivatives of $\phi$ at the point $t_k$:

$$J_k = \frac{\partial \phi}{\partial t}(t_k) = \begin{bmatrix} 1/t_{k,2} & 0 & -t_{k,0}/t_{k,2}^2 \\ 0 & 1/t_{k,2} & -t_{k,1}/t_{k,2}^2 \\ t_{k,0}/l & t_{k,1}/l & t_{k,2}/l \end{bmatrix},$$

(7.14)

where $l = \left\| (t_{k,0}, t_{k,1}, t_{k,2})^T \right\|$.

The local affine approximation of the compound mapping from source to ray space $x = m_k(u)$ is given by the concatenation of $t = \phi(u)$ and $x = \phi_k(t)$:

$$x = m_k(u) = \phi_k(\phi(u)) = J_k Wu + x_k + J_k \cdot (d - t_k).$$

(7.15)
We substitute \( \mathbf{u} = \mathbf{m}_k^{-1}(\mathbf{x}) \) in Equation 7.11 and apply Equation 2.17 to map the Gaussian reconstruction kernels to ray space, yielding the desired expression for \( r'_k(\mathbf{x}) \):

\[
r'_k(\mathbf{x}) = g_{\mathbf{R}'_k}(\mathbf{m}_k^{-1}(\mathbf{x}) - \mathbf{u}_k) = \frac{1}{|W^{-1}J_k|}g_{\mathbf{R}'_k}(\mathbf{x} - \mathbf{m}_k(\mathbf{u}))
\]

(7.16)

where \( \mathbf{R}'_k \) is the variance matrix in ray coordinates. According to Equation 2.17, \( \mathbf{R}'_k \) is given by:

\[
\mathbf{R}'_k = J_k W \mathbf{V}_k \mathbf{W}^T J_k^T
\]

(7.17)

Note that for uniform or rectilinear datasets, the product \( \mathbf{WV}_k \mathbf{W}^T \) has to be computed only once per frame, since \( \mathbf{V}_k \) is the same for all voxels and \( \mathbf{W} \) changes only from frame to frame. Since the Jacobian is different for each voxel position, \( \mathbf{R}'_k \) has to be recalculated for each voxel, requiring two \( 3 \times 3 \) matrix multiplications \( \mathbf{R}'_k = J_k(\mathbf{WV}_k \mathbf{W}^T)J_k^T \). In the case of curvilinear or irregular volumes, each reconstruction kernel has an individual variance matrix \( \mathbf{R}_k \). Our method efficiently handles this situation, requiring only one additional \( 3 \times 3 \) matrix multiplication, i.e., \( \mathbf{R}'_k = (J_k \mathbf{W}) \mathbf{V}_k(\mathbf{W}^T J_k^T) \). In contrast, previous techniques [136, 78] cope with elliptical kernels by computing their projected extents in screen space and then establishing a mapping to a circular footprint table. However, this procedure is computationally expensive. It leads to a bad approximation of the integral of the reconstruction kernel as pointed out in [95, 127].

As illustrated in Figure 7.4, the local affine mapping is exact only for the ray passing through \( \mathbf{t}_k \) or \( \mathbf{x}_k \), respectively. The figure is exaggerated to show the non-linear effects in the exact mapping. The affine mapping essentially approximates the perspective projection with an oblique orthographic projection. Therefore, parallel lines are preserved, and approximation errors grow with increasing ray divergence. However, the errors do not lead to visual artifacts in general [95], since the fan of rays intersecting a reconstruction kernel has a small opening angle due to the local support of the reconstruction kernels.

A common approach of performing splatting with perspective projection is to map the footprint function onto a footprint polygon in camera space in a first step. In the next step, the footprint polygon is projected to screen space and rasterized, resulting in the so-called footprint image. As mentioned in [95], however, this requires significant computational effort. In contrast, our framework efficiently performs perspective projection by mapping the volume to ray space, which requires only the computation of the Jacobian and two \( 3 \times 3 \) matrix multiplications. For spherical reconstruction kernels, these matrix operations can be further optimized as shown in Section 7.2.

**Integration and Band-Limiting.** We integrate the Gaussian reconstruction kernel of Equation 7.16 according to Equation 7.6, resulting in a Gaussian footprint function \( q_k \):

\[
q_k(\mathbf{x}) = \int_{\mathbb{R}} \frac{1}{|W^{-1}J_k|}g_{\mathbf{R}'_k}(\mathbf{x} - \mathbf{x}_k, x_2 - x_{k_2})dx_2 = \frac{1}{|W^{-1}J_k|}g_{\mathbf{R}'_k}(\mathbf{x} - \mathbf{x}_k)
\]

(7.18)

where the \( 2 \times 2 \) variance matrix \( \mathbf{R}'_k \) of the footprint function is obtained from \( \mathbf{R}'_k \) by skipping the last row and column, as shown in Section 2.6.2.
Finally, we choose a Gaussian low-pass filter \( h(x) = g_H(x) \), where the variance matrix \( H \in \mathbb{R}^{2 \times 2} \) is typically the identity matrix. With Equation 2.19, we compute the convolution in Equation 7.10, yielding the EWA volume resampling filter, or EWA volume splat:

\[
\rho_k(x) = (p_k \otimes h)(x) = c_k o_k \left( \frac{1}{W^{-1} J_k^{-1}} (g_{R_k'} \otimes g_H)(x - x_k) \right) \\
= c_k o_k \left( \frac{1}{W^{-1} J_k^{-1}} g_{R_k'} H (x - x_k) \right) 
\]  

(7.19)

7.2 IMPLEMENTATION

We implemented a volume rendering algorithm based on the EWA splatting equation. Our implementation is embedded in the VTK (visualization toolkit) framework [120]. We did not optimize our code for rendering speed. We use a sheet buffer to first accumulate splats from planes in the volume that are most parallel to the projection plane [136]. In a second step, the final image is computed by compositing the sheets back to front. Shading is performed using the gradient estimation functionality provided by VTK and the Phong illumination model.

We summarize the main steps that are required to compute the EWA splat for each voxel in Equation 7.5: First, we compute the camera coordinates \( t_k \) of the current voxel \( k \) by applying the viewing transformation to the voxel center. Using \( t_k \), we then evaluate the Jacobian \( J_k \) as given in Equation 7.14. In line 4, we transform the Gaussian reconstruction kernel from object to ray space. This transformation is implemented by Equation 7.17, and it results in the \( 3 \times 3 \) variance matrix \( R_k' \) of the Gaussian in ray space. Remember that \( W \) is the rotational part of the viewing transfor-
mation, hence it is typically orthonormal. Additionally, for spherical kernels, $R_k$ is the identity matrix. In this case, evaluation of Equation 7.17 can be simplified significantly. Next, we project the voxel center from camera space to the screen by performing a perspective division on $t_k$. This yields the 2D screen coordinates $x_k$. Now we are ready to set up the resampling filter $\rho_k$ according to Equation 7.19. Its variance matrix is derived from $R_k$ by omitting the third row and column, and adding a $2 \times 2$ identity matrix for the low-pass filter. We compute the determinants $1/|J_k^{-1}|$ and $1/|W^{-1}|$ that are used as normalization factors, and we evaluate the shading model yielding the emission coefficient $c_k$. Finally we rasterize the Gaussian filter in an axis aligned bounding box as described in Equation 5.3.1.

7.3 RESULTS AND COMPARISON

In Figure 7.6 we compare the properties of the EWA volume resampling filter (top row) with the antialiasing filter proposed by Swan et al. [127] (bottom row). The EWA filter adapts its shape to the low-pass filter under minification and to the reconstruction filter under magnification. It provides a smooth transition between minification and magnification, allowing for non-uniform scaling of the reconstruction filter under anisotropic minification-magnification. In the bottom row, we show the filter shapes resulting from uniformly scaling the reconstruction kernel to avoid aliasing as proposed in [127]. Essentially, the reconstruction kernel is enlarged such that its minor radius is at least as long as the minor radius of the low-pass filter. For spherical reconstruction kernels, or circular footprint functions, this is basically equivalent to the EWA resampling filter. However, for elliptical footprint functions, uniform scaling leads to overly blurred images in the direction of the major axis of the ellipse.

We further illustrate the difference between our method and Swan's approach in Figure 7.7. The images on the left were rendered with EWA volume splats, those on the right with Swan's uniformly scaled kernels. We used rectangular zebra textures with $x \times y$-dimensions of $1024 \times 512$ voxels (in the first row), and $1024 \times 256$ (in the second row), and mapped the textures to a square. This leads to elliptical reconstruction kernels with a ratio between the length of the major and minor radii of 2 to 1 and 4 to 1, respectively. The output images have a resolution of $400 \times 150$ pixels. Clearly, the EWA filter produces a crisper image and at the same time does not exhibit aliasing artifacts. As the ratio between the major and minor radii of the reconstruction kernels increases, the difference to Swan's method becomes more pronounced. For strongly anisotropic kernels, Swan's uniform scaling produces excessively blurred images, as
shown on the right in Figure 7.7. Each frame took approximately 6 seconds to render on an 866 MHz PIII processor.

FIGURE 7.6 Comparison between the EWA resampling filter and Swans’s reconstruction kernel.

FIGURE 7.7 Comparison between EWA volume splatting (left) and Swan et al. (right). Top row: 1024 × 512 × 3 volume texture. Bottom row: 1024 × 256 × 3 volume texture.
In Figure 7.8, we compare EWA splatting using surface reconstruction kernels on the left to volume kernels on the right. The texture size is $512 \times 512$ in $x$ and $y$ direction, and the output image resolution is $500 \times 312$ pixels. Typically, the perspective projection of a spherical kernel is almost a circle. Therefore, rendering with volume kernels does not exhibit anisotropic texture filtering and produces textures that are slightly too blurry, similar to isotropic texture filters such as trilinear mip-mapping. On the other hand, splatting surface kernels is equivalent to EWA texture filtering. Circular surface kernels are mapped to ellipses, which results in high image quality because of anisotropic filtering.

In Figure 7.9, we show volume renderings of the UNC CT scan of a human head ($256 \times 256 \times 225$ voxels), the UNC engine ($256 \times 256 \times 110$ voxels). The images illustrate that our algorithm correctly renders semitransparent objects as well.

We can also combine EWA volume and EWA surface splatting, as illustrated in Figure 7.10. While the foot of the visible woman dataset ($152 \times 261 \times 220$ voxels) is rendered using volume splatting, the textured plane in the background is rendered using surface splatting. The images in Figures 7.9 and 7.10 have a resolution of $512 \times 512$. 

---

**FIGURE 7.8** EWA volume splatting versus EWA surface splatting: $512 \times 512 \times 3$ volume texture

**FIGURE 7.9** Semi-transparent objects rendered using EWA volume splatting: (a) UNC head, (b) engine, (c) visible woman foot dataset. [see Color Plate 4 on page 159]
300 \times 300 \text{ pixels, and it took approximately 11 seconds to render them on an 866 MHz PIII processor.}
CHAPTER 8

EDITING PARAMETERIZED POINT-SAMPLED SURFACES

While we have focused on the representation and rendering of point-sampled surfaces in the previous parts of this thesis, we describe a framework for interactive point-based surface editing in this chapter. Raw point data is often acquired by 3D scanning systems. But despite recent advances of such scanning technology, this data usually has to be modified to obtain satisfactory 3D objects for computer graphics applications. Also, the user may want to change the shape or the appearance of scanned 3D objects. The techniques described in this chapter complement the point-based representation and rendering algorithms presented before. They enable us to build a point-based processing pipeline ranging from data acquisition to surface editing and rendering. At no stage in this pipeline it is necessary to reconstruct a triangulated surface. We believe that such a system, entirely based on points, can considerably simplify the process of 3D content creation, e.g., for internet e-commerce applications.

Our approach generalizes techniques for editing 2D digital images to make them amenable to processing 3D point-sampled surfaces. Where previous 3D modeling, painting, or sculpting systems are either based on polynomials [4], triangle meshes [1, 113], implicits [105, 106], or images [97], our system is founded on irregular 3D points as a powerful and versatile surface representation. We generalize 2D picture elements (pixels) towards 3D surface elements (surfels [128, 107]) to combine 3D surface editing with the simplicity and effectiveness of 2D photo processing.

Our framework is based on two key components, namely surface parameterization and surface resampling. These are at the core of most of the surface editing tools. For instance, surface texturing or carving both demand a flexible parameterization of the point cloud. In addition, points discretize geometry and appearance attributes at the same rate. For example, surface detail embossing with a high resolution depth map can lead to aliasing and hence requires the adaptation of the surface sampling rate.
The rest of this chapter is organized as follows: In Section 8.1, we introduce a conceptual framework to edit point-sampled 3D models that allows us to implement a variety of tools to alter the geometry and appearance of irregular point-sampled geometry. The scope of possible operations goes well beyond the functionality of 2D photo editing systems. We implemented painting, texturing, embossing, and filtering; however, our system is easily extendable towards new operations. In Chapter 8.2, we present a method for distortion minimal parameterization of point clouds, extending prior work on triangle meshes [71]. The algorithm allows for user constraints that can be adjusted interactively to control the parameterization. In Section 8.3, we explain how to use the parameterization by projection approach, which is also at the core of EWA surface splatting (Chapter 5), for editing point-sampled surfaces. Independent of the parameterization method, we describe surface resampling and the surface editing operations in Section 8.4 and Section 8.5, respectively. Finally, we in Section 8.6 present results produced using a number of editing operations, including painting, texturing, carving, displacement mapping, and filtering.

The goal of our system is to explore the usability of point primitives for surface editing. As such, it constitutes an alternative to conventional polygonal mesh or spline based approaches. Since our algorithms are based on \(k\)-nearest neighbor search, input data with a substantial amount of noise or highly irregular sampling distribution, e.g., as acquired by multiple merged range scans, can lead to instabilities. In these cases, the raw scans have to be resampled to a clean point cloud, for example using distance fields [25]. Further, the techniques described in this chapter are suitable for editing fine scale surface detail, rather than for applying large deformations or CSG operations. However, such methods have been developed by Pauly [101, 103].

8.1 SYSTEM OVERVIEW

Our editing framework for point-sampled surfaces is inspired by the way image editing is performed in 2D photo editing systems. To give an overview of our approach we first describe a typical 2D image editing operation on an abstract level. Then we explain how these concepts can be generalized to point-sampled surface editing.

We consider a 2D digital image as a set of samples on a uniform 2D grid, the samples representing image attributes such as color or transparency. While image editing operations are usually performed directly on the sampled image, it is possible to compute a continuous representation using a reconstruction filter whenever necessary (see also Chapter 2). We describe a general image editing operation as a function of an original image and a brush image, where the brush image is used as a general tool to modify the original image. For example, the brush image may be interpreted as a paint brush or a discrete filter according to the desired operation.

The editing operation proceeds as follows: First, we establish a common frame of reference for the image and the brush by specifying a parameter mapping. In 2D image editing, the mapping usually translates the center of the brush to the pixel at the current mouse position. Next, we define a common sampling grid for the image and the brush, such that there is a one-to-one correspondence between the discrete samples. This requires a resampling operation to be performed either on the image or the brush: First, the continuous image function (or brush function, respectively) is reconstructed and then the continuous function is sampled on the common grid. Finally, an editing oper-
ator combines the image samples with the brush samples using the one-to-one correspondence established before.

We now generalize this procedure to irregular point-sampled surfaces, as illustrated in Figure 8.1.

Essentially, we do this by replacing the discrete image by a parameterized point-based surface, as introduced in Chapter 4.

The main challenges that arise with this approach concern the parameterization, resampling, and editing of point-sampled surfaces. Photo editing deals with planar images that have a trivial parameterization. However, defining a parameterization of a manifold surface is much more involved. In our system, the user interactively selects a subset of the surface, which we call a patch. We compute a parameterization of the patch that assigns parameter coordinates to each point in the patch and then apply the
editing operation on the parameterized patch. We provide two methods for selecting and parameterizing patches: In Section 8.3, we present a simple and efficient method of parameterizing surface patches by projection. In general, such a mapping leads to distortions that cannot be avoided completely. In Section 8.2, we present a method that minimizes these distortions while letting the user intuitively control the mapping.

Since images are usually discretized on a uniform grid, signal processing methods can be applied directly for filtering, resampling, etc. (see Chapter 2). However, the sampling distribution of surfaces is in general nonuniform, requiring alternative methods for reconstruction and sampling such as proposed in Chapter 3. Our interactive editing system is based on the parameterized surface representation introduced in Chapter 4. This representation allows us to reconstruct a smooth surface directly from the point cloud, which can be resampled (i.e., evaluated) at any desired parameter location.

After the one-to-one correspondence between surface and brush samples has been established, editing operations can be performed directly on the samples, as in 2D photo editing. It is straightforward to implement surface painting, texturing, or texture filtering. However, since we are dealing with parameterized texture and geometry, the scope of operations is much broader. Additional editing operators include carving, displacement mapping, and geometry filtering.

### 8.2 MINIMUM DISTORTION PARAMETERIZATION

The procedure to compute a minimum distortion parameterization of a surface is illustrated in Figure 8.2. The user first marks an arbitrary surface patch using a dedicated selection tool and specifies a set of corresponding feature points on the surface (Figure 8.2a) and in the 2D parameter domain (Figure 8.2b). Then she initiates the constrained minimum distortion parameterization algorithm that attempts to match the feature points while simultaneously minimizing the distortion of the parameter mapping. These constraints are expressed as an objective function, which is formulated mathematically in Section 8.2.1. The objective function is discretized as described in Section 8.2.2, which results in a linear equation system. This system is solved using a nested iteration approach presented in Section 8.2.3. Based on the framework of Chapter 4, the parameterized points are then used to define a smooth surface as described in Section 8.2.4. Finally, the user performs a series of editing operations on the parameterized patch, such as filtering or texture mapping (Figure 8.2c).

#### 8.2.1 Objective Function

Our algorithm for computing minimum distortion parameterizations of point-sampled objects is based on an objective function similar to Levy’s method for polygonal meshes [71]. In contrast to this approach, we derive a discrete formulation for surfaces represented by scattered points without requiring any tessellation (Section 8.2.2).

Let us denote a continuous parameterized surface patch by \( s_p \). The patch is defined by a one-to-one mapping \( P : \Omega \rightarrow s_p \in \mathbb{R}^3 \) which for each point \( x = (x_0, x_1)^T \) in \( \Omega = [0, 1] \times [0, 1] \) represents a point \( p = (p_0, p_1, p_2)^T \) on the surface:
The mapping $P$ describes a parameterization of the surface, with $X = P^{-1}$ its inverse. Our method computes a parameterization that optimally adapts to the geometry of the surface, i.e., minimizes metric distortions. Additionally, the user is able to specify a set $M$ of point correspondences between points on the surface $p_j \in \mathbb{IR}^3$ and points in the parameter domain $f_j \in \mathbb{IR}^2$, $j \in M$, to control the mapping. This can be expressed as the following objective function:

$$C(P) = \sum_{j \in M} \varepsilon \|P(f_j) - p_j\|^2 + \int_\Omega \gamma(x) dx,$$

where $\gamma(x) = \int_\theta \left( \frac{\partial^2}{\partial r^2} P_x(\theta, r) \right)^2 d\theta,$

and $P_x(\theta, r) = P \begin{bmatrix} x + r \cos(\theta) \\ x + r \sin(\theta) \end{bmatrix}.$

The first term in Equation 8.2 represents the fitting error as the sum of the squared deviations from the user specified data points. The second term measures the smoothness, or distortion, of the parameterization. At each surface point, $\gamma(x)$ measures the squared curvature of the parameterization in each radial direction using a local polar reparameterization $P_x(\theta, r)$. If $\gamma(x)$ is zero, the parameterization at $x$ is a so called polar geodesic map, which preserves arc length in each radial direction [100, 134].
With the parameter $\varepsilon$, the user additionally controls the relative weight of the data fitting error and the smoothness constraint. The desired parameterization $P$ can be obtained by computing the minimum of the functional of Equation 8.2. We now describe how to set up and minimize Equation 8.2 in the discrete case.

### 8.2.2 Discrete formulation

Given a set of distinct points $\{p_i\}$ defining the surface patch, our goal is to assign to each point $p_i$ a point $x_i$ in the parameter domain, such that the objective function is minimized. In other words, we are solving for the unknown discrete mapping $X: p_i \rightarrow x_i$ and hence we reformulate Equation 8.2 by substituting the unknown $X$ for $P$. Moreover, we assume that the parameterization is piecewise linear, thus the second derivative of $X$ is not defined at the points $p_i$ in general. As an approximation, we discretize the smoothness criterion by computing at each point $p_i$ the squared difference of the first derivatives along a set of normal sections. This yields the following objective function $\tilde{C}(X)$:

$$\tilde{C}(X) = \sum_{j \in M} \varepsilon \| f_j - X(p_j) \|^2 + \sum_{i = 1}^{n} \sum_{j \in N_i} \left( \frac{\partial X(p_i)}{\partial v_j} \frac{\partial X(p_i)}{\partial \bar{v}_j} \right)^2,$$

(8.5)

where $n$ is the number of points in the patch, $N_i$ specifies the set of normal sections, and $v_j$ and $\bar{v}_j$ are unit vectors on the surface given by the normal section.

**FIGURE 8.3** Computing directional derivatives using normal sections. (a) Situation in 3D, (b) local frame in 2D.

**Directional Derivatives.** We compute the directional derivatives $\frac{\partial X(p_j)}{\partial v_j}$ and $\frac{\partial X(p_j)}{\partial \bar{v}_j}$ in Equation 8.5 as illustrated in Figure 8.3a: At each point $p_j$, we collect a set $N'_i = \{i_1 \ldots i_k\}$ containing the indices of its $k$ nearest neighbors, typically $k = 9$. For each neighbor $p_j, j \in N'_i$, we determine the plane $P$ defining the normal section, which is given by the normal $n_i$ at $p_i$ and the vector $v_j = p_j - p_i$. We then choose the two points $p_\alpha$ and $p_\beta, \alpha, \beta \in N_i$, such that the angles between $v_\alpha = p_\alpha - p_i$ and $v_\beta = p_\beta - p_i$ and the plane $P$ are minimal, while the angles between $v_j$ and $v_\alpha$, and between $v_j$ and $v_\beta$ are bigger than 90 degrees. Otherwise,
the normal section crosses the boundary of the patch, hence we ignore it. This procedure is sufficient to handle patches with boundaries. Next, we compute the direction \( \mathbf{v}_j \) of the intersection line of the plane \( P \) and the plane given by \( \mathbf{p}_i, \mathbf{p}_\alpha, \) and \( \mathbf{p}_\beta \) (see Figure 8.3).

Assuming a piecewise linear mapping \( X \) between \( \mathbf{p}_i \) and \( \mathbf{p}_j \), the directional derivative at \( \mathbf{p}_i \) along \( \mathbf{v}_j \) is simply

\[
\frac{\partial}{\partial \mathbf{v}_j} X(\mathbf{p}_i) = \frac{\mathbf{x}_i - \mathbf{x}_j}{\|\mathbf{v}_j\|},
\]

where we used the notation \( \mathbf{x}_i = X(\mathbf{p}_i) \).

Likewise, we compute the derivative along \( \mathbf{v}_j \) by assuming a piecewise linear mapping on the triangle defined by the points \( \mathbf{p}_\alpha, \mathbf{p}_\beta, \mathbf{p}_\gamma \) (similar to [71, 72]). Let us denote the coordinates of a local 2D frame in the plane defined by this triangle by \( \mathbf{t} = (t_0, t_1) \) (Figure 8.3b). The local frame is given by \( \mathbf{v}_j \) and a vector \( \mathbf{y} = \mathbf{n}_i, \mathbf{v}_j \times \mathbf{v}_j \), where \( \mathbf{n}_i, \mathbf{v}_j \) is the unit normal of the triangle \( \mathbf{p}_\alpha, \mathbf{p}_\beta, \mathbf{p}_\gamma \). We express the 3D triangle vertices \( \mathbf{p}_\alpha, \mathbf{p}_\beta, \mathbf{p}_\gamma \) in local 2D coordinates \( \mathbf{p}_\alpha, \mathbf{p}_\beta, \mathbf{p}_\gamma \) by solving the equations

\[
[\mathbf{v}_j] \mathbf{p}_i + \mathbf{p}_i = \mathbf{p}_i, \quad [\mathbf{v}_j] \mathbf{p}_i + \mathbf{p}_i = \mathbf{p}_i, \quad \text{and} \quad [\mathbf{v}_j] \mathbf{p}_i + \mathbf{p}_i = \mathbf{p}_i, \tag{8.7}
\]

where \( [\mathbf{v}_j] \) is a \( 3 \times 2 \) matrix. Note that \( \mathbf{p}_i = (0, 0) \) is the origin of the local frame.

We now define the parameterization on the triangle as a linear interpolation in local coordinates. Linear interpolation on a triangle is most easily expressed in barycentric coordinates \( r \) and \( s \), i.e.,

\[
\mathbf{x}(r, s) = r \cdot \mathbf{x}_i + s \cdot \mathbf{x}_\alpha + (1 - r - s) \cdot \mathbf{x}_\beta = [\mathbf{x}_i - \mathbf{x}_\beta, \mathbf{x}_\alpha - \mathbf{x}_\beta] \cdot \begin{bmatrix} r \\ s \end{bmatrix} + \mathbf{x}_\beta, \tag{8.8}
\]

where the relation between barycentric coordinates \( r \) and \( s \) and local coordinates \( \mathbf{t} \) is given by

\[
\mathbf{t} = \xi^{-1}(r, s) = r \cdot \mathbf{p}_i + s \cdot \mathbf{p}_\alpha + (1 - r - s) \cdot \mathbf{p}_\beta = [\mathbf{p}_i - \mathbf{p}_\beta, \mathbf{p}_\alpha - \mathbf{p}_\beta] \cdot \begin{bmatrix} r \\ s \end{bmatrix} + \mathbf{p}_\beta. \tag{8.9}
\]

By substituting \( [r, s] = \xi(t_0, t_1) \) in Equation 8.8, we get the parameterization in local coordinates:

\[
\mathbf{x}(t_0, t_1) = [\mathbf{x}_i - \mathbf{x}_\beta, \mathbf{x}_\alpha - \mathbf{x}_\beta] \cdot \xi(t_0, t_1) + \mathbf{x}_\beta. \tag{8.10}
\]

To simplify the notation, we introduce the matrix \( \mathbf{C} = [\mathbf{p}_i - \mathbf{p}_\beta, \mathbf{p}_\alpha - \mathbf{p}_\beta] \), hence

\[
\begin{bmatrix} r \\ s \end{bmatrix} = \xi(t_0, t_1) = \mathbf{C} \cdot (\mathbf{t} - \mathbf{p}_\beta). \tag{8.11}
\]

The partial derivative \( \partial \mathbf{x}(t_0, t_1)/(\partial t_0) \) is then
\[ \frac{\partial x(t_0, t_1)}{\partial t_0} = [x_i - x_{p_\beta}, x_{\alpha} - x_{p_\beta}] \cdot \begin{bmatrix} C_{0,0} \\ C_{1,0} \end{bmatrix}. \]  

(8.12)

Hence for \( \partial X(p_j)/(\partial \tilde{v}_j) \), we have the linear expression
\[ \frac{\partial}{\partial \tilde{v}_j} X(p_j) = C_{0,0} \cdot x_i + C_{1,0} \cdot x_{\alpha} - (C_{0,0} + C_{1,0}) \cdot x_{p_\beta}. \]  

(8.13)

In contrast to Floater’s shape preserving weights [38, 37], our method can be used as an extrapolator, since we do not enforce the coefficients of Equation 8.13 to be a convex combination. As a consequence, we do not have to specify a convex boundary. Still, our method has the reproduction property: If all points lie in a plane and at least three or more points obeying an affine mapping are given as fitting constraints, the resulting parameterization will be an affine mapping, too. Moreover, we do not need to construct a local triangulation at each point as in [38] to establish our constraints. Note that the parameterization is not guaranteed to be bijective. It is rather left to the user to select a suitable patch and appropriate point correspondences to obtain the desired mapping.

### 8.2.3 Nested Iteration Least Squares Solver

The discrete objective function of Equation 8.5 is now a sum of squared linear relations of the general form
\[ \tilde{C}(X) = \sum_{j=1}^{n} \left( b_j - \sum_{i=1}^{n} a_{j,i} \cdot x_i \right)^2 = \| b - A x \|^2, \]  

(8.14)

where \( x \) is a vector of all unknowns \( x_i = (x_{i,0}, x_{i,1})^T \) and the coefficients \( a_{j,i} \) result from Equation 8.6 and Equation 8.13. The elements \( b_j = (b_{j,0}, b_{j,1})^T \) of the constant vector \( b \) are zero, except for points that are associated with a fitting constraint (see also Equation 8.5):
\[ b_j = \begin{cases} f_j & \text{if } j \in M \\ (0, 0)^T & \text{otherwise} \end{cases}. \]  

(8.15)

We compute this linear least squares problem by solving the normal equations
\[ A^T A x = A^T b. \]  

(8.16)

Since the system matrix \( A^T A \) is symmetric positive definite by construction, we use a conjugate gradient approach [7] to solve Equation 8.16.

The convergence of such iterative solvers can be further accelerated by efficient multilevel techniques. To this end, we designed a hierarchical strategy as illustrated in Figure 8.4. In a top-down pass, we contract the system by recursively clustering the unknowns \( x \). The clustering is driven by the spatial proximity of the corresponding surface points \( p \), similar as proposed in [102], and each cluster yields one unknown on the current level. In a bottom-up pass, we solve Equation 8.16 starting with the coarsest level. The solution is then prolonged by assigning it as an initial value to the next
higher resolution level. This process, which is known as nested iteration in the literature [15], is repeated recursively up to the original resolution.

Figure 8.2 depicts an example of our parameterization technique. We first apply the minimum distortion parameterization on a complex surface patch, controlling the mapping by specifying a set of corresponding feature points. Finally, we perform a texture mapping operation.

8.2.4 Defining a Continuous, Smooth Surface

In this section, we describe how to make use of the global parameter values \( \mathbf{x}_i \) at each point to define a continuous surface in the form of Equation 4.3 (Section 4.2). Recall that independent of global parameter values defined at each point, Equation 4.3 relies on local fitting functions and reconstruction filters in local parameter domains \( \mathbf{u}_i \) around each point. The continuous surface is constructed by bringing all local parameterizations into one common frame of reference using mappings \( \phi_i : \mathbf{u}_i \rightarrow \mathbf{x} \). Note that to compute the fitting functions and reconstruction kernels in the local parameter domains, we determined local coordinates \( \mathbf{u}_i \) of a number of samples \( S_j, j \in N_i \), where \( N_i \) contains the indices of the \( k \)-nearest neighbors around \( S_j \).

Hence the idea behind computing the mappings \( \phi_i \) is to choose them such that the local parameter coordinates \( \mathbf{u}_i \) of a sample \( S_j, j \in N_i \), match its global parameter coordinates \( \mathbf{x}_j \) computed in Section 8.2.1, i.e., \( \mathbf{x}_j \approx \phi_i(\mathbf{u}_j), j \in N_i \). In our method, we restrict the mappings \( \phi_i \) to be affine, and we compute them by minimizing the sum of squared errors

\[
\sum_{j \in N_i} (\mathbf{x}_j - \phi_i(\mathbf{u}_j))^2,
\]

which is again a linear least squares problem. Since both the local parameterization and the patch parameterization are smooth in a neighborhood \( N_i \), local affine mappings \( \phi_i \) provide a sufficient approximation and have lead to good results in our system. With the mappings \( \phi_i \) in place, we express the local fitting functions \( p_i^A(\mathbf{u}_j) \) and reconstruction filters \( r_i(\mathbf{u}_i) \) in the patch parameter domain as \( p_i^A(\mathbf{x}) = p_i^A(\phi_i^{-1}(\mathbf{x})) \) and \( r_i(\mathbf{x}) = r_i(\phi_i^{-1}(\mathbf{x})) \) respectively. The parameterized smooth surface is then defined as in Equation 4.3 (Section 4.2).
Instead of applying this two-step approach with a local fit followed by parameter matching, we could also directly compute the fitting functions in the patch parameter domain. However, our scheme is more efficient and versatile when the global parameterization changes often. We then have to recompute the parameter matching only, instead of recomputing the fitting functions and the reconstruction kernels. We also may use the same local parameterizations for different applications, i.e., rendering (Chapter 5), or texture mapping using the minimum distortion parameterization described above.

### 8.3 PARAMETERIZATION BY PROJECTION

Parameterization by projection is very similar to the rendering algorithm introduced in Chapter 5. Essentially, the parameterization is defined by using the image plane as the 2D parameter domain. In our interactive system, parameterization by projection proceeds as illustrated in Figure 8.5: The user centers the brush image at the desired location on the surface, which is typically the current mouse position (Figure 8.5a). We then align the brush with the tangent plane at this point and orthogonally project the surface points onto the brush image plane. The parameterized patch is given as the set of points that have their projection inside the brush image. To each of these points, we assign the resulting coordinates in the brush image plane as its parameter coordinates (Figure 8.5b). This process combines the patch selection and parameterization step using a simple projection. It is therefore suitable only for small patches with limited curvature. Finally we apply the pre-selected editing operation, as illustrated in Figure 8.5c.

![Figure 8.5](image)

**FIGURE 8.5** Projection by parameterization: (a) Interactively aligning the brush image to the tangent plane, (b) patch selection and parameterization by orthogonal projection, (c) editing operation, e.g., texture mapping.

More formally, we map local parameter coordinates \( \mathbf{u}_i \) to the patch parameter domain \( \mathbf{x} \) using an orthogonal projection \( \mathbf{x} = \Pi_i(\mathbf{u}_i) \). Using the mappings \( \Pi_i \), we can then project the fitting functions \( p_i^j(\mathbf{u}_i) \) and reconstruction filters \( r_i^j(\mathbf{u}_i) \) to the patch parameter domain, i.e., \( p_i^j(\mathbf{x}) = p_i^j(\Pi_i^{-1}(\mathbf{x})) \) and \( r_i^j(\mathbf{x}) = r_i(\Pi_i^{-1}(\mathbf{x})) \). We finally reconstruct the continuous surface in image space, similar to Section 5.1.
8.4 RESAMPLING

Resampling consists of two separate steps: First, the reconstruction step should provide a smooth, accurate approximation of the continuous surface, i.e., of all its shape and appearance attributes. To avoid aliasing while preserving surface detail, the actual sampling step should then use a suitable sampling grid and properly band-limit the surface before evaluating it at the new sampling locations.

After selecting a patch, parameterizing the samples and determining the mapping from the local parameterizations to the patch parameter domain using the methods of either Section 8.2 or Section 8.3, reconstruction is performed by means of Equation 4.3 (Section 4.2). Before actually sampling the surface, we need to determine a suitable sampling grid in the patch parameter domain. Usually, the grid is chosen such that it preserves the detail contained in both the original surface and the brush image. Alternatively, one could compute an optimal sampling distribution given a fixed number of samples.

In our interactive surface editing system, we provide two different sampling strategies. The first approach uses the original surface points as the resampling grid, effectively resampling the brush at the surface samples. The advantage of this method is that we do not have to insert any new surface points, and there is no information loss in the surface samples due to resampling. On the other hand, in many operations such as texture mapping we want to resample the surface at the sampling distribution of the brush that represents the texture, avoiding any loss in texture quality. Since the brush is often sampled on a regular grid, the evaluation of the surface function can be optimized using incremental calculations, e.g., for evaluating the Gaussian weight functions and polynomial fitting functions similar as in Section 5.3.

To avoid aliasing artifacts when evaluating the surface function at the resampling grid we have to properly band-limit the continuous function before sampling. In a regular signal processing framework, band-limiting is performed by convolving the function with a suitable low-pass filter. Our approach is inspired by signal processing, approximating this procedure with irregular sampling distributions, however. We approximate the convolution by replacing the reconstruction filters in Equation 4.3 (Section 4.2) by band-limited reconstruction filters, or resampling filters, which leads to Equation 4.7. Note that this resampling procedure is applied to all surface attributes, such as color, position, normal, etc.

We illustrate resampling during surface editing in Figure 8.6, where we mapped a high resolution texture containing a text pattern onto a sphere. In Figure 8.6a, we sampled the texture at the original surface points without band-limiting the texture. As a consequence, the texture detail is lost and aliasing appears. In Figure 8.6b, we pre-filtered the texture before sampling, such that aliasing is avoided. The close-ups for both figures visualize the sampling pattern of the sphere. Finally, we resampled the surface at the resolution of the texture in Figure 8.6c. Here, texture detail is fully preserved. The close-up to the right shows the new sampling pattern of the resampled surface patch, which is much finer than the original pattern.
8.5 SURFACE EDITING

The resampling technique of Section 8.4 provides a one-to-one correspondence between samples of the surface and samples of the brush. We can thus combine corresponding samples by applying an editing operation such as painting, carving, or displacement mapping.

**Painting** operations modify appearance attributes by alpha-blending corresponding surface and brush samples. For example, the diffuse color of a surface sample can be modified by alpha blending it with the diffuse color of the corresponding brush sample. Similarly, painting can be applied to other attributes such as transparency or material reflectance properties.

We can also apply **normal displacements** to the positions of the surface samples. The new position \( \mathbf{p}_i' \) is given as \( \mathbf{p}_i' = \mathbf{p}_i + \overrightarrow{d}_i \cdot \mathbf{n}_i \), where \( \mathbf{p}_i \) is the original position, \( \overrightarrow{d}_i \) is a scalar coefficient given by the corresponding brush sample, and \( \mathbf{n}_i \) is the surface normal at \( \mathbf{p}_i \). As illustrated in Figure 8.7c, this type of editing operation is particularly suitable for embossing or engraving.

---

**FIGURE 8.6** Resampling during interactive editing: (a) Sampling using the original sampling pattern without filtering the texture. (b) Sampling as in (a), but with pre-filtering the texture. (c) Resampling the surface at the texture resolution. Close-ups emphasize the sampling patterns.
Carving implements a CSG-type operation that subtracts a shape specified by the brush from the surface. The shape of the brush is given by a reference plane and the brush values \( \vec{d}_i \) that are interpreted as a distance from the reference plane. The reference plane is defined by the surface position and normal at a user specified location. The new sample positions are then given by

\[
p_i' = \begin{cases} 
\vec{b}_i + \vec{d}_i \cdot \vec{n} & |\vec{p}_i - \vec{b}_i| < \vec{d}_i \\ 
\vec{p}_i & \text{otherwise}
\end{cases}
\]

where \( \vec{b}_i \) are points on the reference plane corresponding to the brush values \( \vec{d}_i \). Further, \( \vec{n} \) is the reference plane normal, corresponding to the surface normal at the user specified location.

Finally, for filtering operations the brush is interpreted as a discrete convolution matrix. During filtering, the convolution is computed by multiplying and adding up corresponding surface and brush values. We can therefore implement arbitrary discrete linear filters by simply choosing the appropriate filter kernel values. Filters can be applied to any surface attribute, e.g., color, normal or also distance from the reference plane for geometric offset filtering.

It is straightforward to combine the above operations with alpha blending. Alpha blending computes a weighted average of the original surface sample and the modified sample, i.e.,

\[
c_i' = \alpha_i \cdot \tilde{\vec{c}}_i + (1 - \alpha_i) \cdot c_i
\]

Here, \( \alpha_i \) is the alpha value stored in the brush that controls the blending, \( \tilde{\vec{c}}_i \) is the original surface attribute, \( \vec{c}_i \) is the surface attribute after one of the above operations has been applied, and \( c_i' \) is the output value. Alpha blending allows us, e.g., to generate smooth transitions between modified and unaltered surface areas. These editing operations are also described in more detail in [101].

We illustrate the different editing operations in Figure 8.7: Figure 8.7a shows a texturing operation with alpha blending, Figure 8.7b is an example of texture filtering, Figure 8.7c depicts displacement mapping on a sphere, and Figure 8.7d illustrates carving on a rough surface. All these operations can be performed interactively with our system [146], i.e., the user gets immediate visual feedback within tenths of a second (depending on the brush size) on a PC class computer (e.g., Pentium IV processor at 2 GHz).

![Figure 8.7](Link to image)
8.6 RESULTS

We have implemented a point-based surface editing system [146] featuring the techniques described in the previous sections. The system is called “Pointshop3D”, and we have made it available under an open source license [39]. Binary files, source code, and documentation can be downloaded from our web site [59]. The system features an extensible architecture that allows programmers to add their own functionality easily. Further, it includes a splat renderer as described in Chapter 5 to provide immediate visual feedback to the user during editing. On a Pentium IV at 2.0 GHz, it renders approximately 500’000 antialiased splats per second at an output resolution of 512 × 512 pixels.

Figure 8.8 depicts an example of a constrained texture mapping operation with 30 feature points on a model with 218’000 points. In Table 8.1 we summarize timings of the nested iteration solver for different patches with varying sizes on this object, recorded on a Pentium IV at 2 GHz. The data for the textured patch depicted in Figure 8.8b is shown in the third row of Table 8.1. In these examples, we used a hierarchy with three levels and a cluster size of two samples.

The textured surface shown in Figure 8.9 consists of 40’880 points. In this example, we specified 40 feature points in the texture shown (Figure 8.9a) and on the 3D surface patch (Figure 8.9d). The feature points induce strong distortions of the parameterization around the nose and the mouth, as we show using the visualization of the iso-parameter lines in Figure 8.9b. The final result after texturing is shown in Figure 8.9c. Note that here we have chosen a value of \( \varepsilon = 1.0 \) (see Equation 8.5) to enforce the matching of the feature points despite the large deformations. In contrast, we performed the same computations with \( \varepsilon = 0.1 \) in Figure 8.9c and 8.9d. While this leads to smaller distortions of the parameterization (Figure 8.9e), the correspondence of the feature points is less precise (Figure 8.9f).

We also tested the benefits of the nested iteration solver on this dataset. The results are summarized in Figure 8.10, where we varied the number of levels and the cluster size in the clustering hierarchy. The initialization stage includes building up the cluster hierarchy and setting up the equation system by computing the discrete minimum distortion constraints (Section 8.2.2). In the solution stage, we solve the equation system as described in Section 8.2.3. To compute a solution without the nested iteration strategy, initialization of the equation system took 5.05 seconds and the solver required 28.41 seconds, totalling 33.46 seconds. Using the nested iteration strategy, we could

<table>
<thead>
<tr>
<th>Unknowns</th>
<th>Setup</th>
<th>Init</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>58170</td>
<td>6.9 sec.</td>
<td>2.2 sec.</td>
<td>1.8 sec.</td>
</tr>
<tr>
<td>107394</td>
<td>14.9 sec.</td>
<td>8.3 sec.</td>
<td>6.6 sec.</td>
</tr>
<tr>
<td>215628</td>
<td>26.3 sec.</td>
<td>12.1 sec.</td>
<td>7.6 sec.</td>
</tr>
</tbody>
</table>

The table shows the timings for the nested iteration solver. The number of unknowns varies between 58170 and 215628, and the setup times range from 6.9 seconds to 26.3 seconds, with initialization times from 2.2 seconds to 12.1 seconds, and update times from 1.8 seconds to 7.6 seconds.
reduce this to 9.05 seconds (four levels, cluster size ten). On the one hand, the time spent for initialization increases when more levels are used; on the other, this is compensated by the faster convergence of the equation system solver. Moreover, the initialization and the solution stages are both faster with increasing cluster sizes.

A third example for texture mapping is shown in Figure 8.11. Here, we parameterized a patch consisting of 7’913 points (Figure 8.11a) in a model with a total of 25’041 samples. The texture and the associated alpha map in this example have a resolution of 600 × 734 points (Figure 8.11b). Figure 8.11c depicts the result of the texture mapping operation; here, we sampled the texture at the original points of the model, i.e., we did not introduce any new samples. In contrast, we resampled the surface patch at the resolution of the texture in Figure 8.11c. Since the texture is sampled much finer than the original model, most of the texture detail is lost in Figure 8.11c, whereas texture quality is preserved Figure 8.11d. The resampled surface in Figure 8.11d contains
To emphasize the effect of resampling the surface, we visualize the sampling distributions in the close-ups in Figure 8.11c and 8.11d.

To produce Figure 8.12a, we started with a sphere with 114k points and then applied the texture from the moon surface shown in Figure 8.12b. We resampled the sphere at the resolution of the texture, which is $700 \times 700$ pixels. Finally, we applied the distance maps depicted in Figure 8.12c in three carving operations.
FIGURE 8.10 Performance results of the nested iteration solver.
FIGURE 8.11 Parameterizing a surface patch with resampling: (a) visualization of the surface patch, (b) the texture and alpha mask to be applied, (c) texture mapping without resampling the surface patch, (d) texture mapping with resampling.

FIGURE 8.12 Footprints on the moon: (a) carved and textured sphere, (b) texture map applied to the sphere, (c) distance maps used for carving. [see Color Plate 10 on page 162]
In this chapter, we summarize the methods and results presented in the preceding sections of this thesis. Finally, we conclude the thesis by pointing out directions for future work.

9.1 SUMMARY

In this thesis, we have investigated the continuous reconstruction, rendering, and editing of point-sampled surfaces. We have defined point-samples as samples of two-dimensional surfaces in three-dimensional space, including geometric (position, normal) and appearance (reflectance properties) attributes of the surface. Point-sampled surfaces consist of nonuniformly distributed point samples without connectivity information, the points discretizing the surface geometry and appearance at the same rate.

Our work was motivated by the simplicity of this framework that promises several advantages over other surface representations. Modern range scanning devices acquire huge volumes of surface point samples. In contrast to triangle meshes, it is not necessary with point-sampled representations to reconstruct connectivity between the points to process and render the data. Point-sampled surfaces reduce the object representation to the essentials needed for rendering. Further, resampling operations are simple and efficient, since points are not restricted to lie on uniform grids and they do not store explicit neighborhood information. This allows interactive surface editing operations and simplifies the construction of multiresolution hierarchies.

In particular, we have developed and analyzed the following techniques:

**Gaussian destination space resampling filters.** Based on classical signal processing theory, we have described the general concept of resampling filters, which are applied when a discrete signal is mapped from a source to a destination domain. We have expressed resampling filters as a combination of a low-pass filter and a warped reconstruction filter in destination space. Further, we have shown that under affine
mappings, destination space resampling filters are computed by a convolution of the reconstruction and the low-pass filter. We have then analyzed Gaussian filters with respect to their spectral and analytical properties. Gaussian filters allowed us to derive an analytic expression for the resampling filter in destination space.

**Parameterized point-sampled surfaces.** We have discussed nonuniform sampling and reconstruction of two-dimensional signals using a local filtering technique. We have extended this approach to point-sampled surfaces, and we have shown how to define a continuous, smooth surface using a parameterization of the point samples. Further, we have illustrated how certain signal processing operations such as smoothing can be applied to these parameterized, point-sampled surfaces.

**Antialiased splatting algorithms.** Based on the concept of Gaussian resampling filters in destination space, we have derived rendering algorithms using a splatting approach for both surface and volume data.

The rendering algorithm for point-sampled surfaces reconstructs continuous surfaces by using image space as the parameterization domain and discretizes the surface on the pixel grid using Gaussian resampling filters. An extended z-buffer facilitates the rendering of semi-transparent surfaces and edge antialiasing. We also present an adaptation of the algorithm that is amenable to the implementation on current graphics hardware. We compare the antialiasing capabilities of our approach to several other texture filtering techniques, showing that the image quality is superior to popular methods such as bilinear filtering or trilinear mip-mapping. Our software implementation renders up to 500'000 points per second on a 2GHz Pentium IV processor, and the hardware accelerated version achieves up to three million points on a NVidia GeForce4 Ti4400 graphics processor.

Our volume rendering technique is based on the analytic integration of Gaussian reconstruction filters along viewing rays, followed by a convolution of the resulting footprint functions with a Gaussian low-pass filter. A comparison to previous antialiasing approaches for volume rendering shows that this leads to superior image quality. Further, our technique is capable of handling elliptical reconstruction filters, such that curvilinear and irregular volume data sets can be rendered.

**Data structure for point-based rendering.** We have presented a hierarchical data structure called the LDC tree tailored for point-based rendering. By analyzing the sampling properties of LDC trees, we have derived appropriate parameters for reconstruction filters. We have also designed a rendering algorithm based on LDC trees that allows for level-of-detail rendering. Since LDC trees consist of a collection of layered depth images (LDIs), they facilitate efficient projection of points to the image plane using an optimized image warping approach. Further, LDC trees are storage efficient due to their semi-regular structure.

**Interactive surface editing.** We have introduced a framework for interactive editing of point-sampled surfaces, striving to extend two-dimensional photo editing functionality to point-sampled surfaces. The framework builds upon two fundamental components, namely continuous surface reconstruction, based on the concept of parameterized point-sampled surfaces as described above, and surface resampling. These components allow the implementation of various editing operations such as painting, texturing, carving, displacement mapping, and filtering.
9.2 CONCLUSIONS AND DIRECTIONS FOR FUTURE WORK

Point-based methods are still a relatively unexplored research area in the field of computer graphics. Hence, there are many open problems, and we restrict ourselves to pointing out directions for future work that are most related to our work. We see opportunities for future work in the following areas:

**Resampling filters.** By exploiting the unique mathematical properties of Gaussian filters, we have derived analytical expressions for Gaussian destination space resampling filters. These filters proved suitable for our applications because they provide reasonable spectral properties and because they can be computed and evaluated efficiently. However, they have certain disadvantages such as their global support. Therefore, we believe it could be beneficial to investigate other families of filters, e.g., b-spline filters with compact support, for resampling properties.

**Data compression and streaming.** Although the LDC tree is quite memory efficient due to its semi-regular structure, storage requirements could be further reduced by applying lossy data compression schemes. In particular, we think it would be promising to combine the hierarchy of the LDC tree with a multiresolution compression scheme, such as wavelet coding. Since LDIs are sampled on a regular grid, they are amenable to wavelet compression, e.g., as demonstrated in [28]. Care has to be taken to handle gaps in the images, since the layered depth pixels contain different numbers of depth layers in general. Individual depth layers could be treated as masked images similar as in [125]. A wavelet based compression scheme would facilitate the progressive transmission (streaming) of LDC trees, which is a prerequisite for internet based applications (e.g., e-commerce applications).

**Rendering algorithms.** While we have illustrated with our software implementation of EWA surface splatting that point-based rendering is suitable for generating high-quality images of complex objects, its performance is not comparable to modern hardware accelerated triangle rendering pipelines (500,000 points versus millions of triangles per second). One way to reduce this performance gap is to implement surface splatting using such hardware, e.g., as we have done in [112]. The potential for optimizing such implementations keeps growing as graphics hardware becomes more flexible and programmable. Another way would be to build a custom hardware architecture for point-based rendering from scratch. Although we do not foresee fundamental obstacles in doing so, such a venture would undoubtedly require a huge engineering effort.

Our current rendering pipeline focuses on high quality antialiasing, however, it provides only basic local illumination models. It has been demonstrated that it is straightforward to render point-sampled surfaces with shadows [46] or using surface reflectance fields [80]. Further, ray tracing approaches have been proposed to render
reflective and refractive materials [118, 132]. However, it has not been attempted to compute a full global illumination simulation using point-sampled surfaces, providing advanced effects such as diffuse inter-reflections and caustics. We believe that it would be interesting to combine the concept of photon maps [55] with point-sampled surfaces to render such effects.

**Surface editing and modeling.** While the editing operations presented in Chapter 8 are suitable for processing fine surface detail, they do not enable large scale surface modeling. In fact, Pointshop3D has recently been extended by functionality such as free-form deformation and computational solid geometry [103]. However, aiming at a comprehensive software system for modeling point-sampled surfaces, there are a number of open issues: Tools for preprocessing scanned data should be made available, including cleaning and registration of range scans. Automatic resampling algorithms should be provided to fill holes in point-sampled surfaces and to compute optimal sampling distributions during interactive surface processing. Finally, alternative parameterizations, e.g., conformal maps [73], that require less user interaction than the one presented in Section 8.2 could be implemented. Conformal parameterizations would be useful for cut-and-paste editing similar to [10].

**Sampling theory for continuous surface reconstruction.** Our parameterization algorithm that leads to the continuous reconstruction of a point-sampled surface (Section 8.2) is based on $k$-nearest neighbor searching to define the smoothness constraints. However, we have made no attempt to specify under which sampling conditions the $k$-nearest neighbor approach makes sense and when this strategy fails. Although our algorithm worked well with the data sets used in practice, it would be reassuring to have a theoretical justification and well-defined criteria that recommend or refuse the application of $k$-nearest neighbor searching for continuous surface reconstruction (see also [101]).
CHAPTER 2 SIGNAL PROCESSING AND ALIASING IN COMPUTER GRAPHICS

Signal Processing Fundamentals

\[ L \quad \text{linear filter} \]
\[ \delta \quad \text{Dirac delta function} \]
\[ (f \otimes h)(x) \quad \text{convolution of signals } f \text{ and } h \]
\[ \omega \quad \text{angular frequency} \]
\[ F(\omega) \quad \text{frequency domain representation (i.e., spectrum) of a signal } f(x) \]

Frequency Analysis of Aliasing

\[ i(x) \quad \text{impulse train} \]
\[ T \quad \text{sample spacing in the spatial domain} \]
\[ \omega_s \quad \text{sample distance in the frequency domain} \]
\[ a_c(x) \quad \text{continuous input signal} \]
\[ A_c(\omega) \quad \text{spectrum of } a_c(x) \]
\[ a(x) \quad \text{discrete input signal} \]
\[ A(\omega) \quad \text{spectrum of } a(x) \]
\[ h^{\omega_\alpha}(x) \quad \text{ideal low-pass filter with cutoff frequency } \omega_\alpha \text{ in the spatial domain} \]
\[ H^{\omega_\alpha} \quad \text{ideal low-pass filter in the frequency domain} \]
**Antialiasing**

\( g_c(x) \) ..................continuous image signal

\( x_i \) ..........................uniform output sampling grid

\( g(x_i) \) ..................discrete image

\( h(x) \) ..................low-pass filter

\( g'_c(x) \) ..................band-limited image signal

\( \xi_i \) ..........................intermediate, nonuniform sampling positions

\( \gamma_c(x) \) ..................intermediate, continuous image function

\( \gamma'_c(x) \) ..................intermediate, band-limited image function

**Digital Filters for Computer Graphics**

\( |\omega| < |\omega_p| \) ..................pass band

\( |\omega_p| < |\omega| < |\omega_s| \) ......transition band

\( |\omega_s| < |\omega| \) ..................stop band

\( \omega_N \) ..........................Nyquist limit

**Gaussian Filters**

\( g_{\sigma^2}(x) \) ..................one-dimensional Gaussian

\( \sigma^2 \) ..................variance

\( G_{\sigma^2}(\omega) \) ..................spectrum of \( g_{\sigma^2}(x) \)

\( g_N^V(x) \) ..................\( N \)-dimensional Gaussian

\( V \) ..........................\( N \)-dimensional variance

\( G_N^V(\omega) \) ..................spectrum of \( g_N^V(x) \)

**Resampling Filters**

\( u \) ..........................source domain coordinates

\( x \) ..........................destination domain coordinates

\( f(u_k) \) ..................discrete input values at positions \( u_k \)

\( r_k(u) \) ..................source domain reconstruction kernel

\( f_c(u) \) ..................continuous input signal

\( x = m(u) \) ..........................mapping from source to destination domain

\( h(x) \) ..................destination domain low-pass filter kernel

\( \rho_k(x) \) ..................destination domain resampling filter

\( m_k(u) \) ..................local affine approximation of \( m(u) \) at \( u_k \)

\( J_k \) ..........................Jacobian of \( m(u) \) at \( u_k \)

\( r_k'(x) \) ..........................reconstruction kernel warped to destination space

\( h'_k(u) \) ..................low-pass filter kernel warped to source space
\( g_{R_k}(u) \) \text{ Gaussian source domain reconstruction kernel with variance matrix } R_k \\
\( g_{H} (x) \) \text{ Gaussian destination domain low-pass filter kernel with variance matrix } H \\

**CHAPTER 3** \textbf{NONUNIFORM SAMPLING AND RECONSTRUCTION} \\

**Local Filtering** \\
\( f(x) \) \text{ unknown continuous function} \\
\( f(x_i) \) \text{ sampled function values at nonuniform positions } x_i \\
\( \tilde{f}(x) \) \text{ sampled function represented as a sum of Dirac delta impulses} \\
\( g(x) \) \text{ continuous function reconstructed from } \tilde{f}(x) \text{ by local filtering, i.e., using a weighted sum of reconstruction kernels} \\
\( s_i \) \text{ coefficients of the reconstruction kernels defining } g(x) \\
\( L_2 \) \text{ space of square integrable functions} \\
\( \|f\|_{L_2}^2 \) \text{ squared } L_2 \text{ norm of a square integrable function } f \\
\( \langle f, g \rangle \) \text{ inner product of functions } f \text{ and } g \\
\( \delta_{ij} \) \text{ Kronecker delta} \\
\( s \) \text{ vector of coefficients } s_i \\
\( F(s) \) \text{ approximation error} \\
\( M \) \text{ matrix of inner products of reconstruction kernels} \\
\( f \) \text{ vector of inner products of reconstruction kernels and input function} \\

**CHAPTER 4** \textbf{POINT SAMPLED 3D GEOMETRY} \\

**Smooth Surface Reconstruction from Parameterized Point Samples** \\
\( S_i \) \text{ surface sample} \\
\( A \) \text{ set of surface attributes} \\
\( p_i \) \text{ surface position of } S_i \\
\( c_i \) \text{ diffuse surface color of } S_i \\
\( u_j \) \text{ local parameter coordinates around } S_i \\
\( N^i \) \text{ index set of the } k \text{-nearest neighbors of } S_i \\
\( e_i(u_j) \) \text{ reference plane at } S_i \\
\( v_{i,0}, v_{i,1} \) \text{ orthogonal vectors spanning } e_i(u_j) \\
\( u_j^i \) \text{ local coordinates of sample } S_j, j \in N^i
\( p_i^A(u_i) \) polynomial fitting function for attribute \( A \) in the neighborhood of \( S_i \) in local parameter coordinates

\( r_i(u_i) \) reconstruction kernel in local parameter coordinates

\( x \) global parameter coordinates

\( \varphi_i(u_i) \) mapping from local to global coordinates

\( r_i^A(x) \) reconstruction kernel in global coordinates

\( p_i^A(x) \) fitting polynomial in global coordinates

\( s_i^A(x) \) global reconstruction of surface attribute \( A \)

**CHAPTER 6 DATA STRUCTURES FOR POINT-BASED RENDERING**

\( h_0 \) spacing of LDC sampling grid

\( \alpha_{max} \) maximum angle between LDI projection directions

\( \alpha_i(n) \) angle between LDI \( i \) and direction \( n \)

\( S_k \) subset of LDC samples on level \( k \)

\( b \) block resolution

\( q \) apex of visibility cone

\( c_Q \) rendering quality parameter

**CHAPTER 5 EWA SURFACE SPLATTING**

\( p_i^P(u_i) \) local fitting function for surface position

\( p_i^C(u_i) \) local fitting function for diffuse surface color

**Continuous Surface Reconstruction in Image Space**

\( x \) image space coordinates

\( \Pi_i(u_i) \) mappings from local parameter coordinates to image space coordinates

\( \overline{\Pi}_i(u_i) \) local linear approximations of the \( \Pi_i \) at \( u_i^j \)

\( J_i \) Jacobian of \( \Pi_i \)

\( s_p(x) \) reconstructed surface positions

**The Surface Splatting Algorithm**

\( q(x) \) exponent of the Gaussian resampling filter

\( r \) cutoff radius

\([x_{0,min}, x_{0,max}]\), \([x_{1,min}, x_{1,max}]\) bounding box for rasterizing the image space resampling filter

\( W(x) \) accumulated weight of resampling filters

\( z(x_k) \) \( z \) -value at output image pixel \( x_k \)
Floating point \( \varepsilon \) - threshold for blending resampling kernel contributions

**Hardware Accelerated EWA Surface Splatting**

\( s \) - texture coordinates of the rectangle representing the resampling filter

\( d_j \) - texture coordinates of the vertices of the resampling filter rectangle

\( \text{Rot}(\theta), \Lambda \) - rotation and scaling relating \( s \) and \( u_i \)

## CHAPTER 7  EWA VOLUME SPLATTING

**Volume Resampling**

\( x \) - ray space coordinates

\( I_{\lambda}(x) \) - light intensity at wavelength \( \lambda \) at \( x \)

\( f'_c(x) \) - extinction function in ray space

\( c_{\lambda}(x) \) - emission coefficient

\( u \) - object space (source space) coordinates

\( w_k \) - scalar weights

\( r_k(u) \) - volume reconstruction kernels in source space

\( t = \varphi(u) \) - mapping from object space to camera space

\( x = \phi(t) \) - mapping from camera space to ray space

\( q_k(x) \) - integrated reconstruction kernel

\( h(x) \) - low-pass filter kernel

\( c_k \) - constant emission coefficient

\( o_k \) - constant attenuation coefficient

\( \rho_k(x) \) - volume resampling kernel

\( t \) - camera coordinates

\( J_k \) - Jacobian of \( \phi \)

\( x = m_k(u) \) - local affine approximation of the mapping from source (i.e., object) space to ray space

\( V_k \) - variance matrix of the Gaussian volume reconstruction kernel in object space

\( R_k' \) - variance matrix of the Gaussian volume reconstruction kernel in ray space

\( \hat{R}_k' \) - variance matrix of the integrated Gaussian volume reconstruction kernel in image space

\( H \) - variance matrix of the Gaussian low-pass filter in image space
CHAPTER 8  EDITING PARAMETERIZED POINT-SAMPLED SURFACES

Minimum Distortion Parameterization

\( s_p \) .................................. continuous parameterized surface patch

\( x \) .............................. parameter coordinates

\( p \) .......................... point on the surface patch

\( p = P(x) \) .................. parameterization of the surface patch

\( x = X(p) \) ................. inverse of the parameterization

\( p_j \) .......................... feature points on the surface patch

\( f_j \) ............................ feature points in the parameter domain

\( C(P) \) .......................... continuous objective function

\( \tilde{C}(X) \) .................. discretized objective function

\( v_j, \tilde{v}_j \) ..................... unit vectors given by normal sections

\( P_\alpha, P_\beta \) ................. vertices of the triangle in 3D defining the first derivative of the parameterization along \( \tilde{v}_j \)

\( P_i, P_j \) ........................ vertices of a line segment in 3D defining the first derivative of the parameterization along \( v_j \)

\( A \) .............................. matrix of linear constraints

\( b \) .............................. vector representing constraints imposed by feature points

Surface Editing

\( p_i \) .......................... original surface position

\( p_i' \) .......................... surface position after editing

\( d_i \) .......................... scalar coefficient given by brush sample

\( n_i \) .......................... surface normal at \( p_i \)

\( \tilde{b}_i \) ........................ sample points on the reference plane
REFERENCES


[76] D. Lischinski and A. Rappoport. “Image-based rendering for non-diffuse synthetic scenes.” In G. Drettakis and N. Max, editors, Eurographics Workshop on Rendering
REFERENCES


COLOR PLATE 1  Silhouette edges produce infinite frequencies in the output image, leading to jagged edges. (a) Original resolution. (b) Close-up exhibiting the staircase structure of silhouette edges. [see Figure 2.5 on page 16]
COLOR PLATE 2  Test objects for the software implementation of EWA surface splatting: (a) Matterhorn, (b) terrain data, (c) helicopter. The Matterhorn data set in (a) is courtesy of the Swiss Federal Office of Topography. [see Figure 5.16 on page 86]
COLOR PLATE 3  Test objects for the hardware accelerated surface splatting algorithm: (a) chameleon, (b) wasp, (c) salamander, (d) Fiesta. [see Figure 5.17 on page 87]

COLOR PLATE 4  Semi-transparent objects rendered using EWA volume splatting: (a) UNC head, (b) engine, (c) visible woman foot dataset. [see Figure 7.9 on page 117]
COLOR PLATE 5  Combining EWA volume and surface splatting. [see Figure 7.10 on page 118]

COLOR PLATE 6  Minimum distortion parameterization: (a) Patch selection and feature points on the 3D surface patch. (b) Texture map with corresponding feature points. (c) Parameter-ization and texture mapping. [see Figure 8.2 on page 123]
COLOR PLATE 7  Editing operations: (a) Texturing with alpha blending, (b) texture filtering, (c) normal displacement, (d) carving on a rough surface. [see Figure 8.7 on page 131]

COLOR PLATE 8  Interactive texture mapping: (a) input texture with feature points, (b) 3D surface patch with corresponding feature points, (c) visualization of iso-parameter lines on the surface, (d) final result after applying the texture to the parameterized surface. [see Figure 8.8 on page 133]
COLOR PLATE 9  Texture mapping with large distortions: (a) input texture with feature points, (b) visualization of iso-parameter lines with $\varepsilon = 1.0$, (c) final result after applying the texture to the parameterized surface, (d) 3D surface patch with corresponding feature points, (e) visualization of iso-parameter lines with $\varepsilon = 0.1$, (f) final result after applying the texture. [See Figure 8.9 on page 134]

COLOR PLATE 10  Footprints on the moon: (a) carved and textured sphere, (b) texture map applied to the sphere, (c) distance maps used for carving. [See Figure 8.12 on page 136]
Matthias Zwicker

March 18, 1973
Born in Frauenfeld, Switzerland, citizen of Waldkirch, St. Gallen, Switzerland

1993
Matura, Gymnasium Frauenfeld, Frauenfeld, Switzerland
(Swiss high school examination)

1998
Diploma in Computer Science as Dipl. Informatik-Ing. ETH, Swiss Federal Institute of Technology, ETH, Zurich, Switzerland

1998 – 2003
Research and teaching assistant, Computer graphics laboratory, headed by Prof. M. Gross, Institute of Scientific Computing, Swiss Federal Institute of Technology, ETH, Zurich, Switzerland