Diss. ETH No. 16664

Representation and Rendering of Implicit Surfaces

A dissertation submitted to
ETH Zurich

for the Degree of
Doctor of Sciences

presented by
Christian Sigg
Dipl. Rechn. Wiss. ETH Zurich, Switzerland
born 4. March 1977
citizen of Switzerland

accepted on the recommendation of
Markus Gross, examiner
Mark Pauly, co-examiner

2006

Abstract

Implicit surfaces are a versatile representation of closed manifolds for modeling, simulation and rendering. They are defined as the isocontour of a volumetric scalar function, which allows intuitive handling of complex topology. Surfaces deformed by physics-based simulation or modeling operations can undergo large changes including tearing and merging without loosing their consistent definition or producing self-intersections or holes. Implicit surfaces can be rendered directly from their volumetric representation with a ray-casting approach, or they can be converted to a parametric representation like triangle meshes for display.

This thesis focuses on different approaches for representing the scalar function which defines the implicit surface and methods to render the surface directly from those representations with real-time performance. A fundamental concept of the algorithms presented in this thesis is to harness the computational power of consumer graphics hardware. Their highly parallel design provides the working ground for very efficient processing and rendering of implicit surfaces. The main obstacle for such algorithms is to comply with the rendering APIs and hardware restrictions of current generation hardware, which are designed for triangle rasterization.

In computer graphics, arbitrary functions are most commonly discretized as a regular grid of sample values. Sampling and reconstruction of a continuous function have been studied extensively in signal processing theory. One fundamental result that can also be applied to implicit surfaces, specifies the sampling frequency required to faithfully capture fine detail in the input data. For intricate objects, a dense regular grid of sufficient resolution can become challenging in terms of memory requirement and processing power.

However, for the small reconstruction kernels commonly used, most of the samples do not contribute to the actual surface shape, but only serve as a binary inside-outside classification. Adaptive sampling can be used to reduce the sampling density in those regions where accurate reconstruction is not required. The octree provides a semi-regular sampling that can significantly reduce storage requirements without sacrificing accuracy in critical areas containing fine surface detail. The semi-regular structure was employed to develop a hashed octree which can be more efficient in terms of performance and memory footprint than the common pointer-based variant.

To increase numerical stability, the surfaces deformed by the levelset method are represented by a field with constant gradient length. The signed distance

transform computes such a field from a triangle mesh and comes in two different varieties. For random or semi-regular sampling, a kD-tree is used to accelerate the closest triangle search. A slight variation of the tree construction is presented which allows more efficient proximity queries and could be beneficial for many other kD-tree applications. For regular sampling in a neighborhood of the surface, an algorithm based on scan conversion of geometric primitives which bound the Voronoi cells of the triangle mesh has proven to be competitive. The rasterization-hardware of graphics cards provides potential to speed up the process. To employ its full performance, an alternative construction of the bounding volumes is presented which avoids the transfer bottleneck of geometry data.

Reconstruction of a continuous function from a regular set of samples is a fundamental process in computer graphics, for example in texture filtering. A tradeoff between performance and quality has to be made, because higher order reconstruction kernels require larger reconstruction filters. Cubic filters provide sufficient quality for offline rendering, but are usually considered too slow for real-time applications. Building on hardware-accelerated linear filtering, a fast method for cubic B-Spline filtering is presented which evaluates large filter kernels with a moderate amount of texture fetches.

The performance of graphics hardware can be leveraged to perform real-time ray-casting of implicit surfaces directly from their volumetric representation. A two-level hierarchical representation is used to circumvent memory limitations of graphics cards and to perform empty space skipping. By using the higher order reconstruction filters, smooth geometric properties of the isosurface, such as normal and curvature can be extracted directly from the volumetric representation. These local shape descriptors can be used to compute a variety of non-photorealistic rendering effects. Due to the deferred shading pipeline, even complex shading modes can be evaluated in real-time.

Quadratic surfaces are a common rendering primitive and have a very compact implicit definition. However, they are usually tessellated for rendering because graphics cards lack support for non-linear rasterization. A hardware-accelerated rendering method based on the implicit surface definition is presented, which uses a mixture between rasterization and ray-casting. The efficiency of the approach is demonstrated by a molecule renderer which employs smooth shadows and silhouette outlining to improve the spatial perception of the molecular structure.

Kurzfassung

Implizite Flächen sind eine vielfältige Repräsentation von geschlossenen Mannigfaltigkeiten für Modellierung, Simulation und Rendering. Sie sind definiert als Isokontur einer volumetrischen skalaren Funktion, welche eine intuitive Behandlung von komplexen Topologien erlaubt. Flächen, die mittels physikalisch basierter Simulationen oder Modellierungsoperationen deformiert werden, können sich stark verändern – einschliesslich Zerreissen und Zusammenschmelzen –, ohne dabei ihre konsistente Definition zu verlieren oder Selbstdurchdringungen oder Löcher zu produzieren. Implizite Flächen können mit einem Ray-Casting-Ansatz direkt aus der volumetrischen Repräsentation gerendert werden, oder sie können für die Darstellung in eine parametrische Repräsentation wie zum Beispiel Dreiecksnetze überführt werden.

Die vorliegende Dissertation konzentriert sich auf verschiedene Ansätze für die Repräsentation der skalaren Funktion, welche die implizite Fläche definiert, sowie die Methoden, die Flächen direkt aus dieser Repräsentation in Echtzeit darzustellen. Ein grundlegendes Konzept der hier präsentierten Algorithmen ist die Nutzbarmachung der Rechenleistung von Grafikhardware. Ihre hochparallele Architektur ermöglicht die effiziente Bearbeitung und Darstellung von impliziten Flächen. Das grösste Hindernis für die Implementation derartiger Algorithmen stellen die bestehenden APIs und die Restriktionen der heutigen Hardware dar, welche primär für das schnelle Rasterisieren von Dreiecksnetzen entworfen wurde.

In der Computergrafik werden beliebige Funktionen meistens auf einem regulären Gitter von Funktionswerten diskretisiert. Sampling und Rekonstruktion von kontinuierlichen Funktionen sind in der Theorie der Signalverarbeitung extensiv studiert worden. Ein fundamentales Resultat, welches auch für implizite Flächen angewendet werden kann, spezifiziert die Samplingdichte, welche für die wahrheitsgetreue Erfassung feiner Details der Eingabedaten notwendig ist. Bei komplexen Objekten kann ein reguläres Sampling in ausreichender Auflösung jedoch die Grenzen von Arbeitsspeicher und Prozessorleistung überschreiten.

Allerdings hat bei den verbreiteten kompakten Rekonstruktionsfiltern nur eine geringe Anzahl an Samples einen Einfluss auf die Form der Oberfläche, während der grösste Teil lediglich der binären Innen-aussen-Klassifikation dient. In den Regionen, welche keine genaue Rekonstruktion erfordern, kann mittels eines adaptiven Samplings die Samplingdichte reduziert werden. Der Octree bietet

ein semireguläres Sampling, das den Speicherbedarf signifikant verringert, ohne dass kritische Regionen mit feinen Oberflächendetails an Genauigkeit verlieren. Die semireguläre Struktur wurde dabei eingesetzt, um einen gehashten Octree zu implementieren, welcher in Bezug auf den Speicherbedarf und die Leistung effizienter ist als herkömmliche, Pointer-basierte Varianten.

Um die numerische Stabilität zu erhöhen, werden Flächen bei einer Levelset-Deformation mit einer Funktion konstanter Gradientenlänge repräsentiert. Die Distanz-Transformation berechnet ein solches Feld aus einem Dreiecksnetz. Für die Berechnung eines semiregulären Samplings der Distanzfunktion wird in der Regel ein kD-Baum benutzt, um die Suche nach dem nächsten Primitiv im Dreiecksnetz zu beschleunigen. In der vorliegenden Arbeit wird eine Variation dieser Baumkonstruktion präsentiert, welche eine effizientere Nachbarschaftssuche ermöglicht und auch für andere kD-Baum-Anwendungen von Nutzen sein könnte. Für die Evaluation eines regulären Samplings in der Nähe der Oberfläche hat sich ein Algorithmus als effizient erwiesen, der auf der Scan-Konvertierung von geometrischen Primitiven basiert, welche die Voronoi-Zellen des Dreiecksnetzes umschliessen. Die Rasterisierungseinheit von Grafikkarten birgt das Potenzial, diesen Algorithmus zu beschleunigen. Um die Beschränkung der Transferrate zu umgehen, wird ein alternativer Konstruktionsansatz der Hüllgeometrie beschrieben.

Die Rekonstruktion einer kontinuierlichen Funktion aus einem regulären Gitter von Funktionswerten ist ein fundamentaler Prozess in der Computergrafik, zum Beispiel für die Texturfilterung. Dabei muss immer ein Kompromiss zwischen Geschwindigkeit und Qualität eingegangen werden, weil Rekonstruktionsfilter höherer Ordnung einen breiteren Filterkernel benötigen. Kubische Filter bieten zwar ausreichende Qualität für Offline-Rendering, sind aber meistens zu aufwendig für Echtzeit-Anwendungen. Aufbauend auf der Hardware-beschleunigten linearen Filterung wird hier eine schnelle Methode für kubische B-Spline-Filterung vorgestellt, welche die Faltung mit dem Filterkernel mit wenigen Texturzugriffen auswerten kann.

Die Geschwindigkeit von Grafikhardware kann genutzt werden, um ein Echtzeit-Ray-Casting von impliziten Flächen direkt aus der volumetrischen Repräsentation auszuführen. Für die Repräsentation wird eine zweistufige Hierarchie verwendet, um Speicherbeschränkungen von Grafikkarten zu umgehen und leere Regionen beim Samplingprozess zu überspringen. Die Verwendung von kubischen Rekonstruktionsfiltern erlaubt es, glatte geometrische Eigenschaften der Fläche wie Normale und Krümmung direkt aus der volumetrischen Repräsentation zu extrahieren. Diese wichtigen Grössen der lokalen Oberflächenform können für eine Vielzahl von Effekten benutzt werden. Dank einem Deferred-Shading-Ansatz können auch komplexe Effekte in Echtzeit berechnet werden.

Quadratische Flächen sind verbreitete Bausteine von komplexeren geometrischen Objekten und besitzen eine kompakte implizite Definition. Weil Grafikkarten keine nichtlineare Rasterisierung unterstützen, werden sie jedoch für die Darstellung normalerweise zu Dreiecksnetzen tesseliert. Hier wird eine Hardware-

beschleunigte Methode präsentiert, welche basierend auf der impliziten Definition eine Kombination aus Rasterisieren und Ray-Tracing anwendet. Die Effizienz des Ansatzes wird mit einem Algorithmus zur Darstellung von Molekülen demonstriert, welcher weiche Schatten und Konturen benutzt, um den plastischen Eindruck der molekularen Struktur hervorzuheben.

Acknowledgments

I would like to thank Professor Markus Gross for giving me the opportunity to work in such an inspiring and challenging environment and for guiding me through the development of my Ph.D.

Thank you to my examiner, Professor Markus Gross and co-examiner, Professor Mark Pauly for taking their time to read this dissertation.

A special thanks to Steven Assa, Christoph Ramshorn, Wendel Wiggins and Dustin Lister at Schlumberger for the many fruitful discussions about surface representation and for the camaraderie during my time at Schlumberger Research in Cambridge, England.

I am also grateful to all of my colleagues in the Computer Graphics Lab for their support and friendship, in particular my office-mates Martin Wicke and Richard Keiser.

This project has benefited from the work of several students. I would like to extend my thanks to Philipp Schlegel, Michael Sauter, and Oliver Staubli for their contributions.

I would like to express my gratitude to my colleague and collaborator Markus Hadwiger, with whom I enjoy sharing ideas the most. This thesis would not have contained so many nice colorful images without his help.

I would also like to thank my family and my friends who have been a constant source of support and motivation.

Finally, a very big thank-you goes to my wife Kristi for providing a loving, supportive and inspiring environment for me in every moment.

This work has been supported in part by Schlumberger Cambridge Research (SCR).


psycho
bunny

Contents

Abstract iii

Kurzfassung v

Acknowledgments ix

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Contributions . . . 3
- 1.3 Organization . . . 5

2 Fundamentals and State of the Art 7
- 2.1 Implicit Surfaces . . . 7
  - 2.1.1 Definition . . . 8
  - 2.1.2 Signed Distance Fields . . . 8
  - 2.1.3 Differential Surface Properties . . . 10
- 2.2 Representation . . . 11
  - 2.2.1 Analytic Functions . . . 11
  - 2.2.2 Basis Function Approach . . . 12
  - 2.2.3 Sampling Grids . . . 14
- 2.3 Modeling . . . 16
  - 2.3.1 Constructive Solid Geometry . . . 17
  - 2.3.2 Levelset . . . 17
  - 2.3.3 Fast Marching Method . . . 19
- 2.4 Construction and Conversion . . . 20
  - 2.4.1 Surface Reconstruction . . . 21
  - 2.4.2 Signed Distance Transform . . . 21
  - 2.4.3 Sweeping Method . . . 22
- 2.5 Rendering . . . 22
  - 2.5.1 Ray Tracing . . . 23
  - 2.5.2 Triangulation . . . 24
  - 2.5.3 Direct Volume Rendering . . . 25
  - 2.5.4 Quadric Rendering . . . 28

3 Hashed Octree 31
- 3.1 Data Structure . . . 32

3.2 Storage Requirement . . . 34
3.3 Basic Operations . . . 34
3.3.1 Cell Access . . . 34
3.3.2 Neighbor Access . . . 35
3.3.3 Node Access . . . 35
3.4 Construction . . . 36
3.4.1 Subdividing a Cell . . . 37
3.4.2 Cell and Node Cache . . . 37
3.4.3 Blocking . . . 38
3.5 Discussion . . . 38

4 Distance Transform **41**
4.1 Closest Point Acceleration Structure . . . 43
4.1.1 kD-Tree Point Query . . . 43
4.1.2 Construction . . . 45
4.1.3 Discussion . . . 46
4.2 Scan Conversion Based Algorithm . . . 47
4.2.1 Generalized Voronoi Diagram . . . 48
4.2.2 Characteristics/Scan-Conversion Algorithm . . . 48
4.2.3 Prism Scan Conversion Algorithm . . . 50
4.2.4 Hardware Accelerated Scan Conversion . . . 54
4.2.5 Performance Evaluation . . . 57
4.3 Meshless Distance Transform . . . 61
4.3.1 Moving Least Squares . . . 61
4.3.2 Radial Basis Function . . . 62
4.3.3 Partition of Unity . . . 63
4.3.4 Distance Field Approximation . . . 64
4.3.5 Far Field Approximation . . . 65
4.3.6 Discussion . . . 66

5 High Order Reconstruction Filters **69**
5.1 Higher Order Filtering . . . 69
5.2 Fast Recursive Cubic Convolution . . . 71
5.3 Derivative Reconstruction . . . 74
5.4 Applications and Discussion . . . 78

6 Isosurface Ray-Casting **81**
6.1 Pipeline Overview . . . 81
6.2 Hierarchical Representation . . . 85
6.2.1 Empty Space Skipping . . . 85
6.2.2 Brick Caching . . . 86
6.2.3 Adaptive Brick Resolution . . . 87
6.3 Ray-Casting Approach . . . 87
6.3.1 Bounding Geometry Approach . . . 88

6.3.2 Block-Level Ray-Casting . . . 89
6.3.3 Adaptive Sampling . . . 91
6.3.4 Intersection Refinement . . . 92
6.4 Deferred Shading . . . 93
6.4.1 Differential Surface Properties . . . 93
6.4.2 Shading Effects . . . 96
6.4.3 Applications . . . 101
6.4.4 Performance Evaluation and Discussion . . . 102

7 Quadric Rendering **105**
7.1 Overview . . . 106
7.2 Splatting of Quadratic Surfaces . . . 106
7.2.1 Implicit Definition in Homogeneous Coordinates . . . 108
7.2.2 Perspective Projection . . . 109
7.2.3 Bounding Box Computation . . . 110
7.2.4 Ray-Quadric Intersection . . . 111
7.3 Molecule Rendering . . . 113
7.3.1 Deferred Shading . . . 115
7.3.2 Soft Shadow Maps . . . 115
7.3.3 Post Processing Filters . . . 117
7.4 Results . . . 117

8 Conclusion **121**
8.1 Summary of Contributions . . . 121
8.2 Discussion and Future Work . . . 123

Bibliography **125**

A Complete Prism Covering **139**

B Quadric Shader Programs **143**

Copyrights **145**

Curriculum Vitae **147**

Chapter 1

Introduction

This chapter gives an overview of the background and motivation of this thesis, as well as the contribution of this research to the current state of the art in representation and rendering of implicit surfaces. It concludes with an overview in which the organizational structure of the thesis is summarized.

Implicit surfaces have proven to be a versatile representation for a wide spectrum of graphics disciplines, including computer vision, geometric modeling, animation, levelset-simulation, and visualization. Evolving surfaces with complicated and changing topology are handled with simple algorithms while the representation guarantees a continuous, hole-free manifold. Several concrete representations of implicit surfaces have been presented, some of them very general and applicable to many problems, others tailored for a specific task. Like any other type of surface representation, an efficient rendering method is required to complete a successful workflow pipeline.

1.1 Motivation

Finding an appropriate digital representation for real-world objects has challenged many researches and a variety of approaches have been presented throughout recent decades. Different requirements and constraints have lead to a range of representations for different applications. For example, 3D object acquisition devices produce a discrete sampling of the object's surface. However, there is still no sampling theory for manifolds and thus no criterion for sufficiently dense sampling to reconstruct a continuous surface exists. In CAD and digital media production, objects are commonly constructed from higher order surfaces such as NURBS, but are then converted to triangle meshes for further processing and display. In general, triangle meshes are probably the most common data structure for representing surfaces and a significant amount of algorithms from modeling to rendering take advantage of their compact and efficient nature. However, triangle meshes also have a few drawbacks. Fine tessellation is required to overcome the coarse

piecewise linear approximation. Because a triangle mesh is only C^0-continuous, normals and curvature of a tessellated surface are usually interpolated between values estimated at the vertices. With a general mapping from the plane to 3D space, parametric surfaces do not need to be manifolds. Thus, they can contain self-intersections and holes that are not found in surfaces of real-world objects.

The volumetric representation of implicit surfaces, on the other hand, naturally classifies space into inside and outside regions. The interface between the two is inherently a closed manifold. The topological information is contained in the volumetric definition and does not require explicit control as needed for parametric surfaces. Thus, implicits can handle complex and even time-dependent topology without effort and lend themselves to simulate natural phenomena, for example multi-phase flow of splashing water or fire flames. However, the advantages of the volumetric representation also incurs some problems. For one, volumetric representations have a much higher memory requirement than triangle meshes with the equivalent amount of surface detail. With increased memory usage, also more computational power is required to process the surface. In fact, the volumetric representations actually contains an infinite amount of isosurfaces, and all of them are processed concurrently. Finally, efficient rendering is difficult because the surface has to be extracted from the underlying representation for display. The goal of this thesis is to address some of these problems and advance the state of the art in representation and rendering of implicit surfaces.

Most of the algorithms presented harness the computational power and programmability of current generation consumer graphics hardware, also called graphics processing units (GPU). GPUs were introduced in 1996 to speed up the rasterization of textured triangles for computer games. The tremendous growth of the gaming industry during the last decade allowed fast-paced development of new GPUs in an annual cycle from several hardware vendors. In addition to increasing performance at much higher rates than for CPUs, more and more features were included in the chips. First, fixed-function transform and lighting (T&L) and extended texturing modes were incorporated in 1999. Two years later, programmable vertex and pixel shading units were presented and have become increasingly flexible with each generation. Currently, these units can be programmed with high-level shading languages and provide floating point precision. The most current graphics hardware unifies vertex shaders and pixel shaders in one single array of arithmetic-logic units (ALUs). Although a considerable part of the die area is still spent on fixed function features such as rasterization, texture filtering and fragment blending, the development will clearly push future GPUs closer and closer to general stream processors [KDR+02]. Already, graphics hardware is employed to increase performance of a number of algorithms not related to computer graphics, for example direct solvers for linear systems [KW03b] or rigid body physics for games.

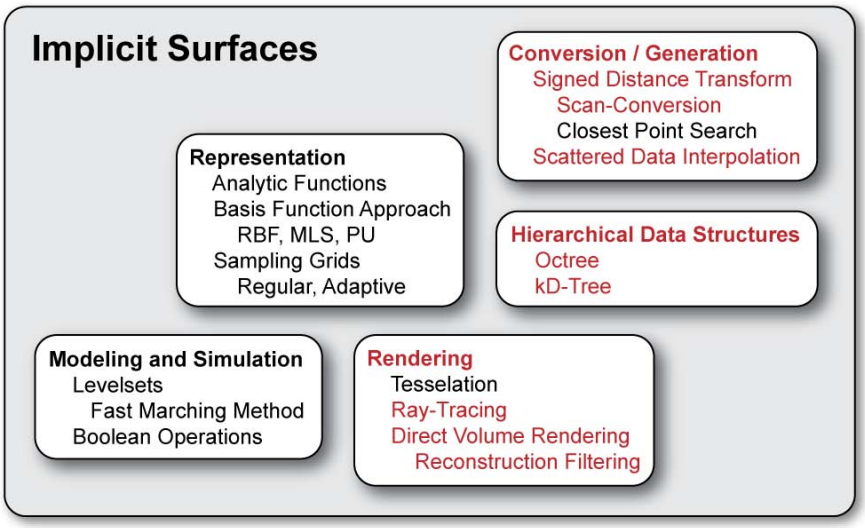


Figure 1.1: Overview of the field of implicit surfaces. The areas of contribution are marked in red.

1.2 Contributions

The field of implicit surfaces (see Figure 1.1) has evolved rapidly over the years as researchers developed compact implicit representations as well as the levelset approach for intuitive modeling of surface evolution. At the same time, performance and flexibility of graphics hardware has improved enormously, outperforming the CPU for many computational tasks which fit the streaming architecture of GPUs. The focus of this thesis was to develop a range of algorithms to represent, process and render implicit surfaces, comprising of the following main contributions:

Hashed octree: Adaptive sampling is a method for adjusting the sampling rate to allow fine local detail without wasting memory in smooth areas. Adaptive distance fields [FPRJ00] use an octree subdivision to perform adaptive sampling of the distance field representing an implicit surface. Octrees are a common semi-regular data structure that usually employ pointers to represent the recursive space subdivision, i.e. a linked tree maps position and size to the corresponding node. However, such sorted associative maps are less efficient than hash maps when no specific order of the nodes is required. Therefore by replacing pointers with hashing smaller and more efficient octrees can be produced.

Hardware accelerated signed distance transform: For the initial state of a levelset simulation, the implicit representation comprising of the distance to the surface is needed as a dense sampling in a close proximity of the surface, called the narrow band. The signed distance transform computes this initial state from a triangle mesh. The characteristics/scan conversion algorithm [Mau00] is a fast variant that computes the distance field by rasterizing a bounding volume of the Voronoi cell of each mesh primitive. An algorithm is presented which performs both rasterization and distance evaluation very efficiently using graphics hard-

ware. An alternative construction method of the bounding polyhedra is derived which reduces the amount of geometry data transferred to the graphics card and deals with incomplete coverage that can occur around saddle vertices.

kD-Tree with node overlaps: For irregular or adaptive sampling of the distance field, the computation is performed per sample instead of per object as for the approach based on scan conversion. To speed up the search of the triangle closest to the sample position, the mesh is stored in a kD-Tree acceleration structure. A variation of the tree construction is presented which allows sibling nodes of the tree to overlap slightly. Thus, the bounding hierarchy can better fit the triangles, which results in more efficient proximity queries and could also be beneficial for many other kD-tree applications.

Distance field extrapolation: When no closed, oriented mesh is available, the distance is extrapolated from discrete samples of the surface. A method that is based on Gaussian-weighted linear fields is presented which produces very smooth interpolation. Similar to compactly supported radial basis function interpolation, a sparse linear system needs to be solved. As in other methods, the extrapolation fidelity degrades when moving away from the surface samples. Thus, the extrapolation is smoothly blended with the distance to the closest point. A simple blend function is derived from quantities already computed by the extrapolation method.

Tri-cubic texture filtering: Reconstruction of a continuous function from a regular sampling is performed by convolving the sample pulses with a filter kernel. General convolution filters can be evaluated in the fragment shader of programmable graphics hardware, but the number of texture lookups required for high-quality cubic filtering significantly degrades performance, especially in higher dimensions. The linear filter modes built into the texture units is employed to considerably reduce the number of texture samples. The method can also be extended to reconstruct continuous first-order and second-order derivatives.

Direct isosurface rendering on graphics hardware: By employing these filters, a direct renderer for implicit surfaces was developed. The surface is reconstructed on-the-fly using a ray-casting approach on graphics hardware. The scalar field is resampled along the viewing rays until a surface intersection has been detected. High-quality intersection positions are achieved by adaptive sampling and iterative intersection refinement. A two-level hierarchical representation of the regular grid is employed to allow empty space skipping and circumvent memory limitations of graphics hardware. Two methods are presented to initiate a fragment program that casts a ray through the volume, which differ in the granularity of empty space skipping and the texture indirection required to access a sample value. Smooth local shape descriptors (e.g. normal and curvature) of the isosurface are extracted from the volumetric definition using the tri-cubic B-Spline filter. They are then used to perform advanced deferred shading using high-quality lighting and non-photorealistic effects in real-time. Sample effects include accessibility shading, curvature color mapping, flow along curvature direction and on-the-fly evaluation of CSG operations.

Rendering of quadratic surface primitives: Some classes of simple implicit shapes can be represented by a compact closed formula instead of sample data. Quadratic surfaces are a common basic building block for more complicated objects and are widely used in many applications. On current graphics hardware, quadrics such as ellipsoids, cylinders and cones can be rendered with a mixture of rasterization and ray-casting. A tight bounding box of the quadric is computed in the vertex program and subsequently rasterized, where the ray-isosurface intersection is computed analytically in the fragment program. Both bounding box and ray hit position can be stated as the root of a bilinear form, corresponding to the implicit surface definition in screen space. Using homogeneous coordinates, the rendering approach presented also supports perspective projections. Quadratic rendering primitives can be mixed seamlessly with standard primitives, such as triangles. The approach is applied to illustrative molecule rendering with high-quality soft shadows and post-processing effects in real-time.

1.3 Organization

Chapter 2 presents fundamentals of implicit surfaces and reviews state of the art techniques. Several representations and methods for constructing them from point samples or triangle meshes are explained. Additionally, approaches for simulation and modeling are introduced and common rendering algorithms are presented. The following four chapters present the main contributions of this thesis. Chapter 3 describes a hashed version of the well known octree. A hardware-accelerated distance transform for triangle meshes is described in Chapter 4. The fast tri-cubic filtering approach presented in Chapter 5 is used in the following chapter to render high-quality isosurfaces with non-photorealistic effects. Chapter 7 shows how hardware-accelerated quadratic surface primitives can improve the spatial perception of molecule rendering. The thesis concludes with a summary of the main results and a discussion of the implications of this work. The appendix includes a proof from Chapter 4 and the source code from Chapter 7 as well as an extensive bibliography.

Chapter 2

Fundamentals and State of the Art

This chapter introduces the methodology of implicit surfaces and discusses state of the art approaches for representation, modeling and rendering.

Implicit surfaces are closed manifolds defined as the isocontour of a scalar function. They provide intuitive handling of complex topology. The signed distance field is a special form of implicit surface definition which provides increased accuracy and stability for the numerical simulation of surface deformation. Different approaches are available to represent the continuous scalar function. Analytic functions provide a compact and efficient representation but can only define a limited set of surface shapes. Basis function approaches are well suited for composing complex surfaces from smoothly blended basic shapes. Surface deformation is modeled with a partial differential equation discretized on a volumetric representation, i.e. a regular grid of function values. Adaptive or sparse sampling has to be employed in order to overcome the storage requirement of volumetric grids. Conversion to an implicit representation can be addressed with several algorithms, depending on the representation of the input surface. For display, the implicit surface needs to be extracted from the representation, either by conversion to a triangle mesh in a pre-process or on-the-fly with a ray-surface intersection search per pixel.

2.1 Implicit Surfaces

Several approaches are available to represent a continuous two-dimensional surface in $\mathbb{R}^3$. *Explicit surfaces* (or *height fields* are defined as a graph over the *xy*-plane; i.e. each point on the surface is expressed as $\mathbf{x} = [x, y, f(x,y)]^T$. The shapes that can be represented as a height field are limited because overhanging ledges cannot be described by a graph. *Parametric surfaces* are defined by a function

$f : \Omega \subset \mathbb{R}^2 \rightarrow \mathbb{R}^3$ which maps the parameter space Ω into 3D space. As surface self-intersections are possible, parametric surfaces like triangle meshes or NURBS surfaces do not generally need to be manifolds. For a surface with complex topology that may even split or merge during some simulated deformation, a consistent definition without self-intersections is difficult to obtain. In such cases, implicit surfaces have been used successfully because their volumetric definition implies a consistent, closed manifold. This chapter will present the theory of these methods and review state of the art techniques to represent, model, deform and render implicit surfaces.

2.1.1 Definition

A general implicit surface $\mathbf{S}$ is defined as the isocontour of a scalar function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$.

$$\mathbf{S} = \{\mathbf{x} \in \mathbb{R}^3 | f(\mathbf{x}) = \rho\} \tag{2.1}$$

The set of points $\mathbf{S}$ is also called *isosurface* [BW97]. The scalar function f actually defines an infinite number of isosurfaces, one for each *isovalue* ρ. Throughout the thesis, the isocontour of $\rho = 0$ will be referred to as the *zero-contour*. Generally, the isocontour of f at level ρ is equivalent to the zero-contour of $f - \rho$. In order for the isosurface to be well defined, it is sufficient that the function f does not have any critical points, i.e. the gradient ∇f must be defined everywhere and must not be zero. In this case, the surface can also be defined as the image of the isovalue under the inverse function.

$$\mathbf{S} = f^{-1}(\rho) \tag{2.2}$$

The isosurface partitions space into two sets, the interior and the exterior of the surface. We follow the convention that the interior is the area where the function is negative, the exterior where the function is positive. All isosurfaces are manifolds. Mathematically, a manifold is characterized by the fact that around every point, there is a neighborhood that is homeomorphic to a disc. Thus, the isosurface does not contain any self-intersections. Also, isosurfaces are always closed and the notion of interior and exterior naturally establishes a consistent orientation.

2.1.2 Signed Distance Fields

A special class of scalar functions to represent the implicit surface has proven to be advantageous for many applications: the gradient is enforced to always have a length of one, which is described by the *Eikonal equation*

$$\|\nabla f\| = 1 \tag{2.3}$$

Together with the boundary condition of the zero-set $f|_S = 0$, the Eikonal equation 2.3 uniquely defines the function f in $\mathbb{R}^3$. Hence, the two are equivalent definitions. At any point in space, f is the Eucledian distance to the closest point on S, with a negative sign on the inside and a positive sign on the outside of the surface. A few intermediate steps are required to understand why the Eikonal equation implies a signed distance field.

Consider two points x and y on a path γ of deepest descent (i.e. the path along the gradient ∇f). The path cannot be shorter than the distance between the two points.

$$\|\mathbf{x}-\mathbf{y}\| \leq |\gamma| = \left| \int_\gamma \nabla f(\mathbf{x}) \cdot \mathrm{d}s \right| = |f(\mathbf{x}) - f(\mathbf{y})| \tag{2.4}$$

On the other hand, the path integral along a straight line ζ connecting the two points x and y is bound by the distance.

$$\|\mathbf{x}-\mathbf{y}\| = |\zeta| = \int_\zeta \underbrace{\|\nabla f(\mathbf{x})\|}_{=1} \|\mathrm{d}s\| \geq \left| \int_\zeta \nabla f(\mathbf{x}) \cdot \mathrm{d}s \right| = |f(\mathbf{x}) - f(\mathbf{y})| \tag{2.5}$$

Hence, γ is equivalent to ζ and $\|x-y\| = |f(x)-f(y)|$. The boundary condition assures that the function is indeed a distance field.

From the triangle equality, it follows that the distance field is a continuous function. Yet, the derivative is only defined almost everywhere. The derivative is not defined at points without unique closest surface points. The set of these points is called the *cut locus* or *medial axis*. The set with two closest surface points is called *skeleton*, the one with three closest points is called *centerline*. Both skeleton and centerline are compact representations of the abstract surface shape and are used for feature description [Blu67], shape recognition, object navigation [WDK01] and animation.

In addition to the cut locus, other properties can be extracted from the distance field. In collision detection and handling for example, penetration depth and direction can be easily extracted from the distance field. The distance field is closely related to the *Voronoi diagram*, which is the partitioning of space with points into *cells* such that each cell consists of the region which is closest to the point associated with the cell. If a surface is divided into *sites* (e.g. the triangles of a mesh), an equivalent *Generalized Voronoi Diagram* (GVD) can be defined.

A good survey of distance fields, their construction and applications, can be found in [JBS06].

2.1.3 Differential Surface Properties

Differential properties of the isosurface can be extracted directly from the implicit representation. The surface normal is the normalized gradient $\mathbf{g} = \nabla f$ of the volume:

$$\mathbf{n} = \frac{\nabla f}{\|\nabla f\|} = \frac{\mathbf{g}}{\|\mathbf{g}\|} \tag{2.6}$$

While the surface normal defines the orientation of the tangent plane, the curvature measures the amount by which a surface deviates from a plane. The curvature of a sphere is the inverse of its radius everywhere: smaller spheres bend more sharply and have higher curvature [Car76]. Thus, curvature is a measure for the size of local surface detail. A minimum sampling frequency for faithful surface and normal reconstruction from discrete representations can also be stated in terms of surface curvature [Gib98, Bær02]. Curvature information of the isosurface can be computed from second order derivatives [MBF92]. The *normal curvature* in any tangent space direction $\mathbf{v}$ is defined as

$$\kappa_n(\mathbf{v}) = \frac{\mathbf{II}(\mathbf{v},\mathbf{v})}{\mathbf{I}(\mathbf{v},\mathbf{v})} = \frac{\mathbf{v}^T(\nabla\mathbf{n})\mathbf{v}}{\|\mathbf{v}\|^2} \tag{2.7}$$

where $\mathbf{I}$ and $\mathbf{II}$ are the first and second fundamental forms [Car76]. The gradient of the normal $\nabla\mathbf{n}$ (also called the *shape matrix* $\mathbf{S}$) is the projection of the normalized Hessian matrix $\mathbf{H} = \nabla\mathbf{g} = \nabla^2 f$ onto the tangent plane.

$$\mathbf{S} = \nabla\mathbf{n} = \nabla\frac{\mathbf{g}}{\|\mathbf{g}\|} = \frac{\mathbf{H}}{\|\mathbf{g}\|} - \frac{\mathbf{g}\overbrace{(\nabla\|\mathbf{g}\|)^T}^{=\mathbf{Hn}}}{\|\mathbf{g}\|^2} = \frac{1}{\|\mathbf{g}\|}(\mathbb{1} - \mathbf{n}\mathbf{n}^T)\mathbf{H} \tag{2.8}$$

The maxima and minima of all normal curvatures are called *principal curvatures*, denoted with κ_1 and κ_2. The corresponding tangent vectors are called *principal curvature directions*. They can be computed by an eigen-analysis of the shape matrix $\mathbf{S}$. Two eigenvalues (eigenvectors) correspond to the principal curvatures (directions). The third eigenvalue is zero, with the normal as the corresponding eigenvector. Other quantities have been derived from the principal curvatures, for example Gaussian curvature ($K = \kappa_1\kappa_2$) or mean curvature ($H = (\kappa_1 + \kappa_2)/2$). A color-coded plot of these quantities on the surface allows very intuitive depiction of the local geometric shape [KWTM03, HKG00]. A color mapping of the principal curvature magnitudes can be used for non-photorealistic volume rendering [RE01] such as ridge and valley lines [IFP95]. Curvature directions can be effectively visualized by advecting dense noise textures [Wij03], which can be done entirely in image space [LJH03] on a per-pixel basis.

2.2 Representation

In section 2.1 the mathematical definition and properties of an implicit surface have been explained. In order to perform geometric modeling or just display the surface on the screen, a concrete representation of the implicit surface is necessary. A wide range of approaches have been presented. Usually, a tradeoff has to be made amongst accuracy, generality, and efficiency, both in terms of memory and computational complexity. For some classes of implicit surfaces, the defining scalar function can be written as an analytic equation, which is a very compact and exact representation. However, the range of surfaces that can be represented is rather limited. Multiple analytic functions can be combined by linear combination or blending to represent more complex functions and surfaces. Several of these basis function approaches have previously been proposed for a wide range of applications. However, the placement of basis functions and choice of appropriate blending weights can be a time-consuming process, regardless of whether they are performed manually or automatically. Thus, deforming the surface for modeling and animation is often difficult with these representations. In digital signal processing, sampling is the most common method for capturing virtually any input data. Both regular and adaptive sampling are common means to represent the scalar function. A reconstruction filter is required to retrieve a continuous definition from the discrete samples. In the following sections, these possible choices of representation will be discussed in greater detail.

2.2.1 Analytic Functions

A set of simple shapes can be defined implicitly by an analytic equation of the position $\mathbf{x} = (x, y, z)^T$. For example, the unit sphere $\mathbf{S}^2$ can be defined as

$$x^2 + y^2 + z^2 - 1 = 0, \tag{2.9}$$

or a plane can be defined by

$$ax + by + cz - d = 0. \tag{2.10}$$

Both examples belong to the class of *algebraic surfaces*, which are defined as the set of roots of a polynomial. The degree of the surface is defined as the degree of the polynomial. Thus, a plane is a linear surface whereas the sphere is a quadratic surface. A wide range of higher order surfaces with interesting properties is known in the mathematics community, i.e. the Klein Bottle or Enneper's Minimal Surface. However, these surfaces often serve the purpose for understanding the features of the underlying function instead of representing objects.

The ultimate goal of an analytic surface representation would be to design a class of simple functions which allow a range of surface shapes controllable through a small set of intuitive parameters. Quadratic surfaces can represent ellipsoids, cylinders, cones and paraboloids in any possible orientation with a mere

set of six variables. Using an implicit definition in homogeneous coordinates, quadratic surfaces can be rendered efficiently on graphics hardware, which will be discussed in Chapter 7. *Superquadrics* [Bar81] provide more flexibility by adding two parameters to control the polynomial exponent. Their intuitive depiction of rotation and anisotropy lend themselves to tensor visualization [Kin04]. While superquadrics are restricted to quadrilateral rotation-symmetry, the *superformula* [GBB03] allows any n-fold symmetry to represent organic shapes like starfish (5-fold) or snowflakes (6-fold).

2.2.2 Basis Function Approach

Analytic representations are designed to describe a surface globally by a single closed formula. Thus, only relatively simple shapes can be represented by analytic functions and only a small set of parameters are available to adjust the shape. More complex shapes are designed as a composition of simpler elements. The implicit definition of the basis elements are composited either by linear combination or smoothly blended across their region of influence.

Blobs

The Blobby Model [Bli82] is inspired by the electron density distribution of molecules. The density of one atom is modeled as a Gaussian function $e^{-\|x\|^2/\sigma^2}$. The model is constructed by superposition of many Gaussians that are scaled and translated to their center. By simple addition, the Gaussian fields (which individually define an implicit sphere) will interact and define a smooth global shape. Unfortunately, evaluating the Blobby Model at a single point requires the summation of all fields because Gaussians extend to infinity. Meta Balls [Gra93] and Soft Objects [WMW86] replace the Gaussian by piecewise polynomial approximation with finite extent. The Blobby model has also been used to fit an implicit surface to range data [Mur91].

RBF

The superposition of translated basis functions is common practice for interpolation and approximation of irregularly sampled data. Thus, it can be employed to gain an implicit definition of a surface that has been sampled by a laser range scanning device [SPOK95, CBC+01]. The general concept is known as *Radial Basis Functions* approach (RBF). In contrast to Blobby Models, where the functions are designed to locally represent the shape of the surface and can be moved around freely, the basis functions in the RBF approach are centered at the sample positions of the input data. A linear system of equations needs to be solved to determine the appropriate linear combination of basis functions in order to interpolate the input

data. The basis function itself is a fixed function of the Eucledian distance to the center point. Common choices include thin-plate ($r^2 \log r$), biharmonic (r) and triharmonic (r^3) splines, which minimize certain bending energy norms resulting in smooth interpolation [CBC+01]. Unfortunately, none of these are compactly supported. The resulting linear system is dense and very expensive to solve. Similar to the original Blobby Model, evaluation also has to consider all basis functions. Compactly supported basis functions produce sparse linear systems and can be evaluated efficiently. However, they do not minimize any variational energy and a multi-scale approach is required to achieve a global definition [OBS03]. More details about the RBF approach can be found in [Buh03].

MLS

Moving least squares [Lev98] is another approach to interpolate or approximate a set of scattered function samples. A polynomial is fitted to the function values in a weighted least squares sense. The important point is that the weights of the least squares fit depend on the point of evaluation, i.e. the error weight degrades the further the distance to the sample position. Instead of using offset points to avoid a non-trivial solution of the implicit surface definition, the MLS surface is defined as the stationary set of an projection operator. This projection operator maps each point in space onto the closest point on the surface. However, the definition of the projection operator is implicit and needs to be approximated by an explicit iterative procedure [Lev03].

MPU Implicits

Multi-level partition of unity implicits is an alternative representation tailored to reconstruct surfaces from very large sets of points [OBA+03]. Instead of radial basis functions, three different types of local surface approximations are used: a quadratic polynomial in three dimensions or over a two-dimensional parameter plane and a piecewise quadratic function to capture sharp features like edges and corners. Those basis functions are fitted locally by analyzing the surface samples and solving a small least-squares problem. As the basis functions do not have any real meaning at distant points, local approximations are blended to a global definition. The space is adaptively subdivided by an octree according to the local surface detail, with one basis function per octree cell. If the basis function cannot adequately represent the samples in one node, the cell is subdivided into eight smaller cells. Neighboring basis functions are blended across the cell boundary to achieve a smooth transition. The blending weights are determined by *partition of unity*, which is a normalized Gaussian around the center of the octree cell, i.e. each blending weight is divided by the sum of all blending weights.

2.2.3 Sampling Grids

A straightforward method to represent arbitrary implicit surfaces is to sample the scalar function on a regular grid. Although many sample values are required to capture fine detail, they can be stored without any overhead in a continuous array that provides fast direct access. Thus, regular grids are a very popular method for rendering algorithms implemented in hardware, such as texturing or volume rendering. For complex shapes, the storage requirement for a regular grid can exceed the memory available on the target platform. The narrow band method or adaptive sampling can mitigate this requirement at the cost of more complex data access. A reconstruction filter is required to retrieve a continuous definition from the discrete samples.

Reconstruction Filters

In signal processing theory, arbitrary continuous data is represented as a discrete set of samples in the spatial domain. After the sampling process, the continuous function is only defined at the discrete sampling locations. To retrieve a continuous definition from the samples, one must perform a process known as *reconstruction* [Hec89]. The reconstructed function is defined as the convolution of a *filter kernel* and the samples, which are interpreted as a series of pulses at the sampling locations.

The definition of the continuous reconstruction can be used to resample the function or derive numerical methods, for example stencils for finite difference operators. The convolution of the sample pulses f_i and the filter kernel h can be written as a convolution sum

$$(f * h)(x) = \sum f_i \cdot h(x - i) \tag{2.11}$$

The choice of the filter kernel is crucial for the quality of the reconstruction. In general, only band limited (i.e. frequency bound) functions can be reconstructed exactly and only if the sampling frequency is twice as high as the highest frequency present in the function. Functions containing higher frequencies than this *Nyquist limit* need to be low-pass filtered before being sampled. Otherwise, high frequencies are captured modulo the Nyquist frequency and show up as low-frequency *aliasing* patterns in the reconstructed function.

The convolution with the filter kernel to reconstruct such a band limited function is equivalent to a modulation in frequency domain [RG75]. For an exact reconstruction, the filter kernel in the frequency domain needs to be 1 below the Nyquist limit and 0 above. The corresponding ideal filter in the spatial domain is the *sinc filter*

$$sinc(x) = \frac{\sin(x)}{x} \tag{2.12}$$

For efficient evaluation, the size of the filter kernel should be relatively small and only a few summands in the convolution sum should be non-zero [RG75]. Unfortunately, the sinc filter has infinite extent and every evaluation needs to take into account all sample values. The box filter on the other hand performs really fast but low-quality nearest-neighbor interpolation. In between those two extremes, a wide range of reconstruction filters have been presented, each with a specific tradeoff between quality and efficiency.

A linear filter interpolates neighboring function samples with straight slopes and is thus the simplest filter providing continuous interpolation. Linear filters are probably the most popular filters in computer graphics, especially in real-time applications. All current graphics hardware natively supports linear filtering of 2D and 3D textures. Higher order filters are usually restricted to software implementations, among them several classes of spline filters [RG75] with varying degree and windowed sinc filters [ML94, MMK+98]. Reconstruction filters are usually divided into *approximating* and *interpolating* classes. Approximating filters like B-Splines do not interpolate the sample values but produce smooth functions that suppress noise in the input data. Catmull-Rom Splines are popular for applications that require value interpolation.

While larger filter kernels can generally achieve a higher reconstruction accuracy, large kernels can have a negative effect on sampled distance functions. The distance function forms *shocks* at the cut locus, which are not band limited. To avoid inaccurate surface reconstruction, the distance between shocks and surface should be larger than the radius of the reconstruction kernel [Bær02]. To achieve this, the ratio between sampling rate and the filter kernel size should at least match the maximum surface curvature.

Narrow Band Approach

Storing a full regular grid at sufficient resolution to capture fine surface detail can exceed the memory available on average workstations. However, only the values in a vicinity of the surface corresponding to the size of the reconstruction kernel actually contribute to the definition of the surface. The narrow band method [AS95] takes advantage of this and only stores a small fraction of all samples in a thin region around the implicit surface as shown in Figure 2.1, right. Usually, a band that is six samples thick is enough to faithfully reconstruct the function and its partial derivative. Outside this band of samples, the value of the function is unknown. A helper data structure associates the sample positions to the sample values, incurring an extra indirection during value access. To keep the overhead per sample low, the narrow band is usually built of small blocks of densely sampled regions. A disadvantage of the narrow band method is that it only allows direct evaluation of the inside-outside test in the sampled region.

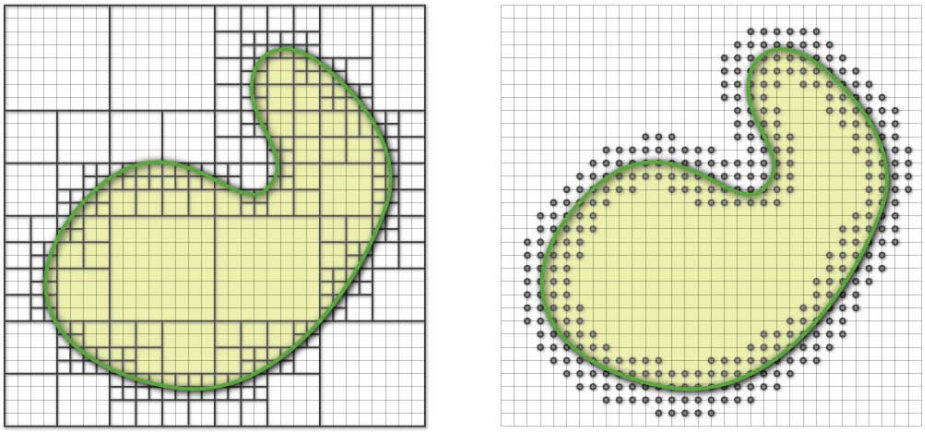


Figure 2.1: Semiregular grids such as the octree (left) and the narrow band (right) allow accurate representation of the isosurface (green) without the overhead of storing a full resolution grid. The octree allows an inside/outside classification of the object (yellow) over the entire domain, but managing and accessing cells in the hierarchy introduces a certain overhead.

Adaptive Sampling

Because the amount of detail is usually spread unevenly across the surface, even the narrow band approach still leaves room for improved memory usage. With adaptive sampling, the sampling rate can be adjusted locally to the frequency spectrum of the surface. In areas that are smooth, only a small number of samples is required to reconstruct the surface, whereas highly curved regions need to be sampled with a large number of samples (see the argument of sampling frequency versus curvature in Section 2.2.3). For the narrow band approach, the sampling rate can be adjusted per block. Adaptively sampled distance fields (ADF [FPRJ00]) store the distance field at the corners of an octree (see Figure 2.1, left and Chapter 3), yielding a very fine grained sampling rate. A cell is subdivided if the trilinear reconstruction does not approximate the real distance function within a certain error bound. For practical reasons, the error is calculated from a small number of test samples, which might miss small features in a region that is generally smooth. In contrast to the narrow band approach, this allows a complete definition of the scalar function for inside/outside tests. Yet, accessing nodes in an octree requires a significant number of memory indirections.

2.3 Modeling

Modeling with implicit surfaces offers several advantages over other surface representations [Set98]. Self intersections cannot occur and the surface can naturally change topology during deformation. The implicit representation of a Constructive Solid Geometry (CSG) operation can be evaluated directly from the implicit

representation of the two operands [MBWB02]. Surface deformation is achieved by solving a partial differential equation (PDE) for the implicit definition known as the *level set* equation [Set98]. The deformation is controlled by the *speed function*, which specifies the speed of the surface in normal direction. If the deformation is monotone (i.e. the speed function does not change sign), the level set equation can be solved with the efficient *fast marching method* discussed in Section 2.3.3.

2.3.1 Constructive Solid Geometry

Constructive Solid Geometry (CSG) provides cut, copy and paste operations for volumetric models [Hof89]. The implicit representation provides a natural point membership classification, i.e. a point is part of the model if the scalar function at that point is smaller than the isovalue. A CSG operation combines two models by evaluating a pointwise Boolean operation (union, difference, intersection) of the membership classification. Table 2.1 lists the implicit representation of the CSG model as a simple function of its operands.

CSG Operation	**Intersection**	**Union**	**Difference**	**Difference**
Boolean Operation	$\mathbf{A} \wedge \mathbf{B}$	$\mathbf{A} \vee \mathbf{B}$	$\mathbf{A} - \mathbf{B}$	$\mathbf{B} - \mathbf{A}$
Scalar Operation	$\min(f_A, f_B)$	$\max(f_A, f_B)$	$\min(f_A, -f_B)$	$\min(-f_A, f_B)$

Table 2.1: Constructive Solid Geometry (CSG) operations for implicit surfaces. Due to the volumetric inside-outside classification, the scalar function defining the result of a CSG operation can be computed directly from the implicit definition of its two operands.

The Boolean operation can either be an intrinsic part of the representation or can be evaluated once the CSG operation has been completed. Volume sculpting systems employ CSG operations and an intuitive interface to create complex solid objects from simpler primitives [GH91,WGG99]. Carving and sawing can be simulated by continuous CSG operation while moving one of the operands [WK95]. Boolean operations produce sharp features at the intersection curve of the two surfaces, which can be captured well by Adaptive Distance Fields [FPRJ00] or blended to a smoother curve [DvOG04]. The implicit surface renderer presented in Chapter 6 supports on-the-fly evaluation of CSG operations.

2.3.2 Levelset

The implicit surface representation allows a very elegant handling of surface deformation. The evolution is formulated as a PDE of the scalar function and discretized on a regular grid using a finite difference scheme. The exact position of the surface is not required to describe the deformation because the implicit representation contains all of the necessary information. Surfaces of complicated

topology can undergo large changes, form holes, split and merge without destroying the closed, consistent, and intersection-free definition.

A particle $\mathbf{x}(t)$ on the isosurface of a time-varying scalar function $f(\mathbf{x},t)$ fulfills the following equation:

$$f(\mathbf{x}(t),t) = 0 \tag{2.13}$$

The derivative with respect to time reveals the fundamental level set equation of motion [OS88]:

$$\frac{\partial f}{\partial t} + \nabla f \cdot \dot{\mathbf{x}} = 0 \tag{2.14}$$

where $\dot{\mathbf{x}} = \mathrm{d}\mathbf{x}/\mathrm{d}t$ denotes the velocity of the particle. The velocity of the surface in its normal direction is determined by the speed function $u = \mathbf{n} \cdot \dot{\mathbf{x}}$. The speed function is a user-defined scalar function which can depend on a number of variables like the position $\mathbf{x}$, surface normal $\mathbf{n}$ or curvature κ. Due to the fact that Equation 2.14 is actually defined everywhere in the computational domain, the definition of the speed function needs to be extended from the surface to the entire volume. However, this is straightforward because normal and curvature can be extracted from the isosurface that passes through the point where the speed function is evaluated. Once the speed function has been defined, the deformation of the surface can be formulated as a PDE:

$$\frac{\partial f}{\partial t} = -\|\nabla f\| \; u(\mathbf{x}, \mathbf{n}, \kappa, ...) \tag{2.15}$$

The solution to the initial value problem of Equation 2.15 can develop shocks, i.e. gradient discontinuities. Thus, a numerically stable discretization needs to incorporate the concepts of weak solutions and entropy conditions [Set98]. The *upwind scheme* can successfully achieve these prerequisites. Essentially, the upwind scheme evaluates derivatives of f as a one-sided difference on the side which produces the larger derivative. As a result, shocks are effectively damped by numerical diffusion. To speed up the numerical process, Equation 2.15 is usually solved only in the narrow band around the surface [AS95]. Reinitialization is needed when the front moves out of the region of defined function values. The set of grid points that belong to the narrow band is reset so that the zero level lies in the center of the band. Alternatively, octree subdivision can be used to discretize the the levelset equation [LGF04].

The levelset approach is a computational approach for a wide range of applications from image processing to path planning. Only the speed function needs to be tailored to the deformation requirements of a specific application. Morphing between two volumetric objects can be implemented as a levelset problem [TO99, BMWM01] as well as surface processing operators like fairing and sharpening [MBWB02]. Levelsets have also been employed to perform physics-based modeling of smoke [LGF04], water and solid objects [GSLF05], as well as fabric draping [BFA02]. Although the automatic handling of changing topology is one of the main advantages of the levelset approach, some applications require the surface to preserve the topology during deformation [HXP03].

The performance of a levelset solver can be improved by employing the computational power of GPUs, for example in volume segmentation [RS01]. Even narrow band levelset computations are possible on GPUs [LKHW03].

For numerical and algorithmic purposes it is advantageous to keep the levelset function close to a signed distance function during the time evolution. It has been observed that the distance property can be maintained to some degree of accuracy. To achieve this, the speed function $u_0 = u|_{f=0}$ defined on the zero-contour has to be extended to the entire domain in a specific fashion. Differentiating the Eikonal equation 2.3 and the levelset equation 2.15 yields a constraint for the speed function:

$$\nabla f \cdot \nabla \left(\frac{\partial f}{\partial t} \right) = 0 \text{ and } \nabla \left(\frac{\partial f}{\partial t} \right) = -\nabla u \quad \Rightarrow \quad \nabla f \cdot \nabla u = 0 \tag{2.16}$$

Thus, gradients of f and u are orthogonal and the speed function is constant along the line perpendicular to the interface [AS99]. In other words, the speed function of the surface is equivalent to the speed function at the closest point on the surface. As a result, the levelset equation 2.15 can be reformulated to preserve the distance field as follows:

$$\frac{\partial f}{\partial t} = -u_0(\mathbf{x} - u\nabla u) \tag{2.17}$$

However, the accuracy decreases with the number of iterations and an occasional reinitialization is unavoidable. This can either be done by recomputing a distance field from the zero-level isosurface (e.g. with the fast marching method presented in the next section) or by performing intermediate iterations with a different speed function that forces the levelset function to attain a distance field [SSO94].

2.3.3 Fast Marching Method

The fast marching method [Set96] is an optimized approach for solving the levelset equation 2.15 for speed functions that are either strictly positive or strictly negative only. In this case, the surface only moves in one direction and the *time of arrival* can serve as implicit surface definition. Thus, the implicit representation is no longer time dependent but stores the whole surface propagation in a single field. Each point stores the time the surface passed through that point and the isosurface of level t corresponds to the surface at time t of the evolution. The arrival time of the surface f is the solution to the Eikonal equation

$$\|\nabla f\| = 1/u \tag{2.18}$$

where $u \geq 0$ is the speed of the surface. The boundary condition of Equation 2.18 is given at the initial surface position, where the function value f is set to zero.

The fast marching method determines the sample values in the order in which the surface passes through them. The upwind scheme propagates information

from samples with smaller time of arrival to larger ones. A thin layer contains sample values that are currently being updated. The values of samples with smaller time of arrival are already known at this point and have been *frozen*. The remaining samples have a larger, but unknown time of arrival. In each iteration, the sample with the smallest value is frozen from the intermediate layer and its value is likewise frozen. The neighbors of the frozen sample with unknown values are then initialized and added to the intermediate layer. As this cycle progresses, more and more grid points are removed and the intermediate layer moves forward, away from the initial front until all grid points are processed. To quickly find the current smallest value, the samples in the intermediate layer are kept in a heap data structure. Sample value are updated with a discretized upwind scheme of Equation 2.18. After a value has changed, the heap must be updated. Hence, the fast marching method is a $O(n \log n)$ algorithm. However, due to the finite difference scheme, the fast marching method is not exact.

In addition to the distance, additional information can be stored in the distance field. Such information can be the vector pointing to the nearest object point, known as the *vector distance transform* [Mul92]. Alternatively, the index of the nearest surface primitive can be attributed to each point, the resulting field is called a complete distance field representation [HCLL01]. By propagating this type of additional information, the fast marching method and similar propagation methods can be turned into exact distance transform algorithms [Nad00, Egg98, Tsa00].

The fast marching method has been successfully applied to several modeling applications. Most importantly, the distance field is the solution of the Eikonal equation with a constant speed of $u = 1$. Thus, all applications involving distance fields, such as computation of medial axis or Voronoi diagrams and levelset reinitialization, can benefit from this method. Morphological operations like erosion and dilation can also be implemented efficiently by using the fast matching method. Erosion removes external parts from a model while dilation adds parts, corresponding to inset and offset surfaces for constant speed functions. A consecutive application of both operations can be used to enlarge cracks and cavities, or remove cavities and smooth spikes [JBS06]. The fast marching method has also proven useful in many other fields, for example computer vision [PS05] or motion path planning [KS98].

2.4 Construction and Conversion

A range of algorithms exist for generating an implicit surface from another representation. They can be classified by the type of representation the input model uses. For point sampled surfaces such as the ones produced by laser range scanners, producing an implicit representation is a good way to retrieve a continuous surface definition. The associated problem can be seen as an interpolation or approximation of the point samples. Parametrized surfaces such as triangle meshes

are converted to a signed distance field in order to model a deformation by the levelset equation, for example based on a physical simulation. If the narrow band method is used, the distance field needs to only be computed in a neighborhood of the surface. For constructive solid geometry operations, a binary inside-outside classification corresponding to the sign of the distance field is sufficient. Finally, propagation approaches are used if the input model is given as a set of voxels or a narrow band representation needs to be extended to a full grid. Apart from the fast marching method discussed in Section 2.3.3, the value propagation can also be carried out in sweeping order.

2.4.1 Surface Reconstruction

Implicit surfaces are well-suited for retrieving a continuous surface definition from a point cloud because no explicit handling of topological information is required. The conversion can be stated as a scattered data interpolation problem [Wen05] because the point cloud is a sampling of the zero-contour solution. Several methods have been presented in the past to compute a continuous function which interpolates or approximates these zero-distance samples. For example, the problem can be formulated as a levelset surface evolution [Whi98, ZO02]. The speed function is designed to attract the isosurface to the sample points while bending energy is minimized. The steady state of the evolution is the desired surface represented on a regular grid. Other implicit representations have been used or even developed specifically for the task of surface reconstruction from a point cloud, such as the Blobby Model [Mur91]. The radial basis function approach can be used to interpolate the distance function at the point cloud and a set of additional offset points. Global basis functions require a large dense linear system to be solved, but construct very smooth surfaces that minimize bending energy [SPOK95, CBC+01]. The linear system for compact basis functions is sparse and can be solved with standard numerical methods, but the solution is not guaranteed to be optimal. Furthermore, a multi-scale approach is required to achieve a global definition [OBS03]. The moving least squares approach [Lev98] can also be used to define an implicit surface as the stationary set of a projection operator [Lev03]. MPU implicits [OBA+03] have been designed to construct an implicit definition from huge sets of surface samples. The space is adaptively subdivided by an octree, and basis functions are fitted to the local sampling. Neighboring basis functions are blended together using the partition of unity approach. Partition of unity blending has also been used to blend first order approximations of the distance field defined at the surface samples [Nie04].

2.4.2 Signed Distance Transform

The signed distance transform [Mau00] computes a distance field on a regular grid from a triangle mesh, for example as the initial state of a levelset computation. The

samples can be computed either in image space or object space, referring to the outer loop of the algorithm, which iterates over all samples or all triangles of the mesh, respectively. Image space methods determine for each sample the corresponding closest point on the triangle mesh. To avoid searching all triangles for the closest point, a spatial search data structure [Sam90] for the triangles is employed. On the other hand, object space methods are based on scan conversion. In this case, the distance field is obtained by scan converting a number of geometric objects related to the triangle mesh and by conditionally overwriting the computed voxel values. Methods based on scan conversion can efficiently compute distance fields in a neighborhood of the surface, e.g. for a narrow band representation. In Chapter 4, a variant of the algorithm presented by Mauch [Mau03] is reimplemented to harness the performance of the rasterization unit of graphics hardware.

2.4.3 Sweeping Method

The fast marching method described in Section 2.3.3 propagates distance values from initial values in the direction of increasing distance until all samples have been processed. With this method, only a thin band of distance values are updated at a time. By using the same upwind scheme, the sweeping methods updates all values concurrently in scan-line order, e.g. sample-per-sample, row-per-row, slice-per-slice. One sweep corresponds to a Gauss-Seidel iteration of the Eikonal equation. The sweeping direction determines the direction of the value propagation and several sweeps in alternating directions are required for the solution to converge. Yet, convergence usually takes no more than one cycle of all possible scan-line directions. Hence, the algorithm has linear complexity in comparison to the superlinear complexity of the fast marching method. However, the fast marching method is considerably faster in typical setups. An extended version which computes exact distance values and an overview of previous methods have been presented in [JQR03].

2.5 Rendering

All surface processing pipelines are comprised of a visualization stage which renders the surface as a two-dimensional image. Rendering isosurfaces represented implicitly by a volume of function samples is an important task in visualization. In medical applications, the volume data is acquired through CT or MRI scans and subsequently analyzed by volume or isosurface rendering. Isosurface rendering is also used to display the operations of surface modeling and animation [MBWB02], or the surface deformation of a levelset simulation [LKHW03]. High-quality rendering at interactive speeds is a major bottleneck, particularly when the isosurface changes over time. The rendering approach for implicit surfaces is fundamentally different than the one for parametric surfaces because the

isosurface has to be extracted from the underlying volumetric representation. Triangulation methods such as Marching Cubes [LC87] do this in a pre-processing step and render the resulting mesh, for example on rasterization hardware. Implicit surfaces can also be rendered from surface samples, which can be generated by a physical-based simulation of particle repulsion [WH94]. Ray-tracing approaches, on the other hand, determine the intersection of a viewing ray per pixel and extract the isosurface directly from the implicit definition [Bar86, Lev88]. Another technique for generating images directly from volume data without any explicit surface extraction is known as volume rendering [DCH88]. Volume rendering employs a transfer function to map the volume data to optical material properties such as color and opacity. The color of a pixel is determined by the volume integral along the viewing ray. Volume rendering can be implemented on programmable graphics hardware with 3D texture support [WE98]. A ray-tracing pipeline for implicit surfaces implemented on graphics hardware will be presented in Chapter 6.

2.5.1 Ray Tracing

Ray tracing a very general method to generate synthetic images. It treats light as set of rays emitting from the light source, traveling through the scene and reflecting at object surfaces and eventually reaching the eye. The process can also be reversed and viewing rays can be shot from the eye point into the scene [Whi80]. To generate photo-realistic images with global illumination effects, a combination of both is usually employed. The light energy is transported from the light source to the surface with a technique like photon mapping [Jen96]. The scene is then rendered with a viewing ray per pixel that gathers the light from the surface elements it hits. In both steps, rays are reflected, scattered and refracted at the surface, which in turn generates new rays. The depth of this recursion determines the final quality of the image but also the required workload.

The most simple and fastest method uses viewing rays only and evaluates a local light model when the ray hits a surface, without generating any additional rays. Thus, the image is generated from *primary rays* only and the image quality is similar to the one achieved by rasterization. Usually, this method is called *ray-casting*. However, we will use the term ray-casting in the domain of volume rendering and refer to ray-tracing even though the scene is rendered from primary rays only.

Finding the closest intersection between a ray and the scene is a central part of any ray-tracing algorithm. For scenes that are composed of small primitives, testing each primitive for an intersection is infeasible and thus, a spatial data structure is employed to speed up the process. The primitives are inserted into a bounding volume hierarchy, which allows to skip intersection tests for all members of a bounding volume that does not intersect the ray. As one is only interested in the first hit, bounding volumes behind an intersection already detected can also be skipped. A bounding volume hierarchy for triangle meshes will be presented in

section 4.1. In Chapter 3, the octree data structure is presented which naturally represents a hierarchy over the regular sampling domain of the implicit surface.

Once an implicit surface is a candidate for an intersection, the ray equation $\mathbf{x}(t) = \mathbf{c} + t\mathbf{v}$ is inserted into the implicit surface definition.

$$f(\mathbf{c} + t\mathbf{v}) = 0 \tag{2.19}$$

where $\mathbf{c}$ is the camera position and $\mathbf{v}$ is the direction of the viewing ray. The intersection closest to $\mathbf{p}$ corresponds to the smallest t to fulfill the constraint 2.19. The equation to solve depends on the type of representation for f. For analytical representation, t is the root of a closed formula which can often be found with a direct approach. Chapter 7 presents a rendering approach for implicit surfaces represented by a quadratic polynomial. For sampled representations, the trilinear interpolation within a grid cell is a third order polynomial along the ray parameter t, which is difficult to solve directly due to limited precision. Also basis function approaches lead to expressions which are too complex to search the root analytically. Thus, a numerical procedure is usually employed, which samples the scalar function along the ray until a zero crossing has been detected. A guarantee that no intersection is missed can still be given under the assumption that the scalar function is Lipschitz continuous [KB89]. However, precise surface intersection and high-quality reconstruction filters are expensive. Hence interactive rates with high-quality or analytic ray-surface intersections and gradients have only been achieved by implementations using multiple CPUs [PSL+98, PPL+99] or clusters [DPH+03]. Different trade-offs have been presented [NMHW02, MKW+04].

2.5.2 Triangulation

Because graphics hardware provides excessive rendering power for polygonal meshes, it is appealing to extract a mesh from the volumetric representation for display. The Marching Cubes algorithm [LC87] extracts a triangular approximation of the isosurface independently for every cell of a regular grid. First, the values at the eight corners of the cell are examined. For each edge that connects two corners with different sign, a vertex is placed at the zero-crossing of the linear interpolation. The sign configuration of all eight corners is then used to lookup the topological configuration of triangles which connects the vertices. After processing each grid cell, the surface is complete. Marching Cubes has been extended to work with adaptive grids such as octrees, where the alignment of triangles of neighboring cells of different size requires special attention [SFYC96]. The Marching Tetrahedron [Blo94, BW97] algorithm extracts a tessellated isosurface from an unstructured mesh. However, the size of the triangles produced by these algorithms is very uneven and sharp features are smoothed out [KBSS01].

Methods that try to overcome these limitations use a dual approach to contouring [JLSW02, VKKM03]. Instead of putting the vertices on the cell edges, the

vertices are placed inside the cells. The vertices of neighboring cells are connected by polygons that intersect the cell edges. Thus, these methods produce meshes that are topologically dual to the ones of Marching Cubes. The position of the vertex inside the cell can be chosen so that the normals of the adjacent triangles approximate the gradient field in a least squares sense [JLSW02]. The increased freedom of vertex position can also be used to reconstruct sharp surface features. Alternatively, one can also run a variation of the Marching Cubes algorithm on the dual grid to achieve similar results [SW04].

The number of cells which need to be tested for triangulation can become too large for interactive changes of the isovalue. The extraction can be accelerated by storing the function ranges of subvolumes in an interval tree [CMM+97]. By using this method, for a given isovalue, only subvolumes which are intersected by the isosurface must be considered.

2.5.3 Direct Volume Rendering

The term direct volume rendering [EHK+06] describes a technique to visualize volumetric data instead of just a single isosurface. The scalar field is interpreted as a density distribution of some media, which can be acquired from medical imaging devices using computer tomography (CT) or magnetic resonance spectroscopy (MRI). A *transfer function* maps the density to optical properties such as color and opacity by classifying different structures in the volume. The volume is then rendered by solving the volume rendering integral, which integrates the color and opacity along each viewing ray corresponding to a pixel of the image. In practice, the volume rendering integral is evaluated by sampling and numerical integration.

For a simple emission-absorption based optical model [Max95], the opacity of participating media is measured in *optical density* D or the more commonly used *extinction coefficient* $\tau = D\,ln(10)$. The *opacity* α is the fraction of light lost by absorption and scattering over a path length L along a viewing ray. *Transmittance* $T = 1 - \alpha$ is the remaining fraction and can be expressed as:

$$T(s) = 10^{-\int_0^s D(t)dt} = e^{-\int_0^s \tau(t)dt} \tag{2.20}$$

A fraction of the color E emitted at a point in the volume is scattered as diffuse color $C = \tau E$. The volume rendering integral describes the diffuse color accumulated along one viewing ray:

$$c = \int C(s)T(s)ds = \int E(s)\tau(s)T(s)ds \tag{2.21}$$

The numerical integration of Equation 2.21 and 2.20 usually assumes piecewise constant or linear values of τ and E. One or two density values are required to perform a lookup into a table which stores the precomputed integrals. The contribution of the individual segments can be accumulated with simple summation (2.21)

and multiplication (2.20). Because the transfer function may be discontinuous, the assumption of piecewise constant density can lead to severe sampling artifacts. This is especially a problem for isosurface rendering, where the transfer function consists of a Dirac function at the isovalue. Piecewise linear methods also known as *pre-integration* lead to much better results with minimal overhead [EKE01].

Multi-dimensional transfer functions allow to emphasize important areas in the volume. For example, the gradient magnitude can be used to favor sharp interfaces between different densities over uniform regions [Lev88]. The gradient direction can also be used to evaluate a lighting model. Transfer functions in the domain of principal curvature magnitudes are helpful for depicting the local shape of the isosurface or for performing non-photorealistic rendering [HKG00, KWTM03, HSS+05]. However, the lookup table for multi-dimensional transfer functions can become very large if the dimensions are not separable or pre-integration has been used.

Most volume rendering algorithms work on a dense regular grid of function samples. Thus, the numerical evaluation of the volume rendering integral involves a resampling process. The choice of the resampling locations has a great impact on image quality and the efficiency of the algorithm. An overview and evaluation of different methods has been presented in [MHB+00]. The most simple one, called *ray-casting*, resamples the volume along each viewing ray with a constant sampling rate. A reconstruction filter is applied at each sampling position to obtain the resampled value, which is then mapped to color and opacity, and composited along the ray.

Filtering of regular data can be carried out efficiently by texture mapping hardware and thus, most algorithms for volume rendering employ graphics hardware to achieve interactive rendering speed. Texture filtering is performed for each fragment that is generated by rasterizing geometric primitives, e.g. triangles. Although volume rendering does not render a triangular surface per se, a *proxy geometry* is required to specify the resampling positions and perform color accumulation. If the sample volume is stored as a 3D texture, one can compute view-aligned *texture slices* that cut the volume parallel to the viewing plane at a regular interval [EKE01]. Texture coordinates specified at the vertices and interpolated across the slice are used to sample the volume at the corresponding locations. The texture hardware performs an automatic trilinear interpolation of the eight closest texture values. Lookup into the transfer function table, pre-integration and evaluation of a lighting model can all be performed in the fragment shader of programmable graphics hardware.

Even though 3D textures are supported by all current graphics cards, filtering 3D textures is generally slower than filtering 2D textures because of the more complex trilinear filter. In volume renderers which employ 2D textures [WE98], the proxy geometry is aligned with the sample grid. All polygons are parallel to one side of the volume and mapped with one 2D texture of sample values. Thus, only hardware-native bi-linear interpolation is required to reconstruct the volume at the sampling positions. However, the sampling can become very uneven and sparse when the proxy geometry is rendered from grazing angles. Thus, three different orientations of volume slices along the major axis are stored on graphics memory. Depending on the viewing angle, the one which is closest to facing the viewing plane is rendered. Unfortunately, there are several drawbacks to this method. First, storing multiple orientations of the volume takes up three times the texture memory. Additionally, switching from one orientation to another abruptly changes the sampling positions, producing visible popping. These errors would be less noticeable if the sampling rate was relatively high. In contrast to the 3D texture approach unfortunately, the number of slices cannot be adjusted to increase the sampling rate because each slice corresponds to one of the 2D textures. Interpolation between neighboring slices can mitigate this [RSEB+00], but then the filtering performance would be even lower than for 3D textures. Finally, there is another problem that applies to all slicing methods under perspective projections but is more severe for 2D textures: The sampling rate is not constant and can thus not generally match the length of the segments for which the integrated color and opacity are stored in a lookup table. Therefore, the values read from the lookup table need to be corrected to account for the local sampling density. *Opacity correction* prevents the most prominent artifacts but is only an approximation [LCN98]. Exact methods are only possible for linear transfer functions [SSP+05].

In ray-casting, the sampling rate can be kept constant or even adjusted locally according to some pre-computed importance volumes [RGW+03]. In addition to hybrid CPU/GPU ray-casting [WS01], hardware accelerated ray-casting has recently become possible by tracking the sampling position in volume space per pixel [KW03a, Gre04]. With support for branching in the fragment program (e.g. loops), the sampling process can even be carried out in a single rendering pass. Such a ray-casting loop will be employed in Chapter 6 to perform real-time isosurface rendering.

In comparison to the size of the CPU's main memory, memory of graphics hardware is usually significantly smaller. Fitting the entire volume into the texture memory can be a challenge for large datasets, which has been addressed by various means of lossy compression [GWGS02,SW03]. Lossless texture packing has been used for adaptive texturing [KE02], improved rendering performance [LMK03], and sparse levelset computations [LKHW03]. Generally, limitations of graphics hardware do not allow sophisticated compression schemes that have been presented for software algorithms, e.g. wavelet compression [GLDK95] .

2.5.4 Quadric Rendering

Quadrics are a higher order rendering primitive, being able to approximate a curved surface with much fewer primitives than a triangle mesh. Models in computer aided design often consist of an assembly of quadratics, such as ellipsoids, cylinders and cones. NVIDIA's first graphics chip (NV1) and Sega Saturn both employed forward-rendered quadratics as basic primitive. However, the wide success of texture mapping and the announcement of Microsoft's DirectX specifications based on polygons in 1995 ended the market interest in these products.

Yet, quadratic surfaces still have their own interface in graphics APIs such as the OpenGL utility library (GLU [WDS99]) and they are still a common class of shapes to be rendered, for example in scientific visualization. Several attempts have been made to provide fast rendering of quadratic surfaces. As a special case of quadrics, spheres lend themselves to sprite rendering, using forward mapping of a precomputed image of a sphere. Depth sprites [MGAK03] additionally read depth offsets from a texture for per-pixel depth corrections. While providing a better approximation than "flat" sprites, this approach is only valid for orthogonal projections and leads, e.g. to incorrect intersection curves between spheres. Moreover, these approaches do not easily generalize to other types of quadrics.

Gumhold uses programmable pixel shaders to compute ray-ellipsoid intersections for tensor field visualizations [Gum03]. This method is most similar to our technique, but computes the bounding box as quadrilateral that contains the silhouette ellipse in world space. While the resulting quadrilateral is a tight enclosure, it requires four times as many vertices per splatted quadric as our method. The rendering method proposed in this thesis uses only a single point primitive for each splatted quadric, requiring an axis-aligned bounding box to determine the respective point size. A combination of sprites and rasterization has been employed to render line tubes [SGS05].

Similarly, early point-rendering approaches, where ellipses rather than ellipsoids are rendered, also use object-space polygons for splat rendering [RPZ02]. However, more efficient methods [BK03] need just one vertex call per splat and employ programmable shaders for their rasterization: the vertex shader computes a screen-space bounding square, and during bounding box rasterization the pixel shader classifies fragments as belonging to the splat or not.

More recently, point-splatting approaches also focused on perspective correctness. The bounding box computation of Zwicker et al. [ZRB$^+$04] is perspectively correct, but computationally expensive and numerically sensitive due to a required matrix inversion. Moreover, the splats' interiors are perspectively distorted in their method. In contrast, the per-pixel ray-casting of Botsch et al. [BSK04] is perspectively correct, but their bounding box computation is only heuristic and might erroneously clip splats. While slightly incorrect bounding boxes are not a problem for mutually overlapping splats representing a single surface, they cause clearly visible, hence unacceptable, artifacts for quadric-based molecule visualizations (see Figure 2.2).

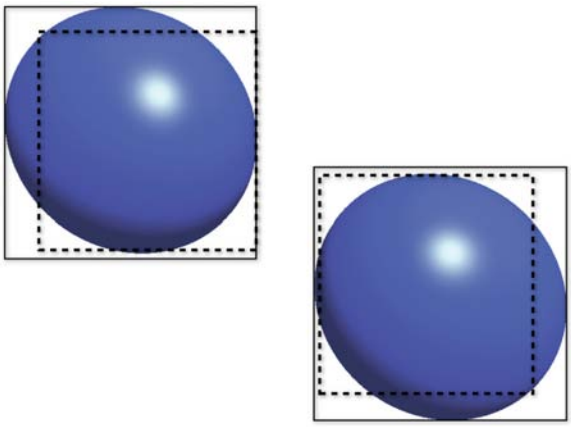


Figure 2.2: Previous heuristic bounding box computations might clip quadrics (dashed box), whereas our homogeneous approach provides perspectively correct results (solid box).

In comparison to [TL04], the approach presented in Section 7 adopts homogeneous coordinates for the implicit definition of the quadric. The resulting bilinear form can be projected into screen space by a simple linear transformation, which enables the robust, efficient, and perspectively correct computation of bounding box and ray intersection, since both can be stated as roots of the homogeneous bilinear form.

To demonstrate one possible application of our method, we show high-quality real-time visualization of molecules. Several standard atomic models are used for the study and dissemination of molecular structure and function. Space-filling representations and ball-and-stick models are among the most common ones. The research community uses a number of free and commercial programs to render these models, each having a specific trade-off between quality and rendering speed. Real-time rendering approaches that can cope with large models typically use hardware assisted triangle rasterization [HDS96], while high-quality images are produced with ray-tracing [DeL02].

Our method exploits programmable graphics hardware to produce real-time visualization of molecules at a quality comparable to off-line rendering methods. We implemented per-pixel evaluation and lighting of quadrics, and integrated soft shadow mapping and silhouette enhancement [MBC02] in order to emphasize the important aspects of the rendered image and to improve spatial perception. A good survey of shadowing techniques can be found in [HLHS03]. High-quality percentage-closer filtering [RSC87] is used to soften shadow edges [Fer05]. The silhouettes of the molecule are outlined by applying an edge detection filter on the normal buffer and the depth buffer.

Chapter 3

Hashed Octree

The sampling resolution of an implicit surface representation naturally limits the amount of detail that can be captured. For example, sheets cannot be thinner than the sampling distance in general. Also, sharp edges and corners beyond the Nyquist limit cannot be well-defined.

Storage requirements of a regular grid often prohibit an accurate resolution capturing even the smallest details of an object. Adaptively sampled distance fields (ADF) change the sampling resolution according to local surface detail [FPRJ00]. The samples are stored in a semi-regular hierarchical data structure, known as an octree (see Figure 3.1). With an adaptive sampling resolution, the storage requirement can be reduced significantly without sacrificing accuracy in critical areas containing fine detail.

This chapter describes an octree that is based on hashing. According to our experiments, the hashed octree can be more efficient and use less memory than the common pointer-based version. Furthermore, a set of optimizations and implementation details for standard operations on an octree designed to represent implicit surfaces will be described.

Because cells in an octree are always subdivided into eight subcells of equal size, it is possible to compute a key (or index) of a cell from its coordinates. Thus, the cell containing a query location can be retrieved efficiently from a hashmap. While this semi-regular configuration is advantageous to represent adaptively sampled distance fields, it is a restriction for building a bounding volume hierarchy of irregular input data. For example, the uniform subdivision is not able to adapt to the triangle coordinates of a triangle mesh. Therefore, the less regular kD-tree is discussed Section 4.1 to perform fast distance queries on triangle meshes. Although some attempts have been made to support higher level data structures on GPUs [LHN05, LKS+06], current architectures are not well suited for implementing octrees or kD-trees because they lack efficient pointer arithmetics. Instead, a relatively simple but effective two-level hierarchy is used in Chapter 6 to store adaptive sampling grids in texture memory of the GPU.

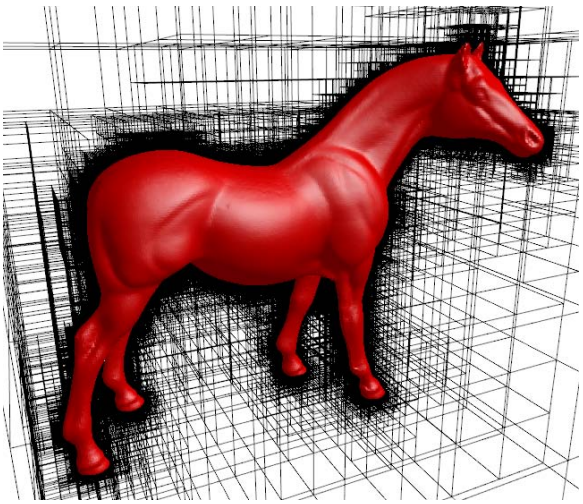


Figure 3.1: Adaptive sampling of the implicit representation. The octree allows to increase the sampling rate in the proximity of the surface to capture fine detail of the model without wasting memory in areas that do not contribute to the surface shape.

3.1 Data Structure

An octree consists of a hierarchy of cubic *cells*. Each cell is potentially split into eight children of equal size. The sampling domain is defined by the largest cell at the root of the hierarchy. Sample values of the scalar function are stored at the corners of the cell, which will be denoted as *nodes*. Within the cell, trilinear interpolation of the corner values is commonly used.

Each cell is defined by its position and size. The size of a cell, corresponding to the edge length, is always half the size of its parent cell. Thus, any cell size is some power of two of the smallest cell. The corresponding exponent is called the *level* of the cell. The smallest cell has level 0, the root cell has level k. This maximum level (also called depth of the tree) has to be fixed in advance. To define a position for each cell, a grid of the smallest cell sizes is laid over the domain of the root cell. Thus, all cell corners will have an integer position in the range $[0, 2^k]^3$. The position of the cell is associated with the position of its upper-left-front corner. Each cell position is aligned with its cell size. This position scheme can be exploited to determine various relations between cell sizes and corner positions using simple bitshift operations.

Fast access to function samples from a cell is required for efficient interpolation and other operations. Therefore, each cell is linked to its eight corner nodes. Note that more than one cell can reference the same node. When a new cell is inserted into the octree, one has to check if any corner node already exists in order to guarantee consistency. Otherwise, the node has to be generated. When a cell is deleted, all corresponding nodes which are not referenced by any other cell have to be deleted too.

The cell hierarchy is usually build as a doubly linked tree. Each cell is linked to its eight children and its parent. A cell can be accessed by traversing down the tree from the root node. Unfortunately, a significant amount of indirections are required to access cells at lower levels. On average, access to a random cell has logarithmic complexity $O(\log n)$, corresponding to the depth of the tree.

On the other hand, hash maps provide element access in constant amount of time. Generally, a hashmap associates *keys* with *values*. To lookup a value given the corresponding key, a hash index of the key is generated to locate the desired value in a linear array. Because multiple keys hash to the same index, the primary location to store a new record might already be occupied. Some collision resolution is then employed to find an empty location in the array to store the value. Later on when looking up the value, the location has to be found again.

We use a hash map to access cells according to their position and size. Collisions are resolved by chaining, which stores multiple records which map to the same hash index in a linked list. The average number of records per slot, called load factor α, is not allowed to grow as more records are inserted into the hash map. Otherwise, access would no longer be possible in amortized constant time.

Less memory is required to store an octree in a hash map instead of in a linked tree. For rather deep, unbalanced octrees used to represent implicit surfaces, hashing also turns out to be faster than linking.

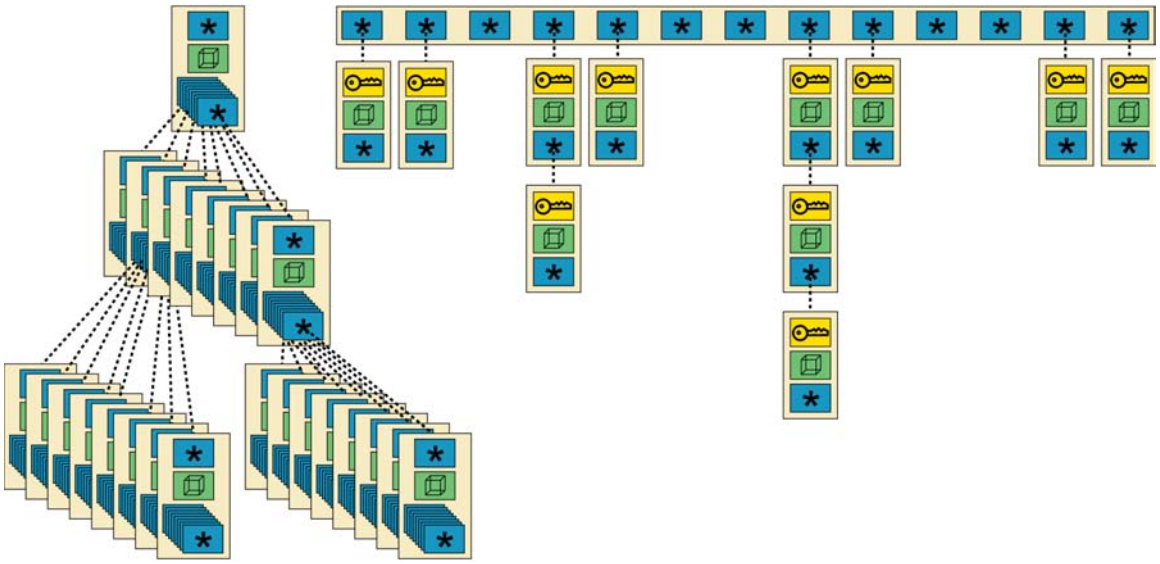


Figure 3.2: Linked and hashed version of an octree. The linked version (left) employs pointers to build a doubly linked tree. The hashed version (right) maps a key consisting of the position and size to the corresponding cell. The hash map uses collision chaining to resolve elements that map to the same hash index.

3.2 Storage Requirement

In order to keep large octrees in memory, the data structure should be as compact as possible. Different data is stored for the two possibilities of linking cells or storing them in hash map. In both versions, the node data and the linking of the nodes from the cells does not change. The hashed version however, does not require any pointers to child or parent cells, because their size and position can always be calculated from the current values and the related cell can then be retrieved from the hash map.

Linked Tree: For a machine using 4 byte memory addresses (32bit), a total of 36 bytes are required to store 9 pointers to parent and child cells.

Hashed Tree: In addition to the actual cell data, each hash records contains a key as well as one pointer for collision chaining (see Figure 3.2). The key consists of position and size of the cell. It can be stored in 13 bytes: the position is stored as a 32bit number per coordinate. Thus, there are at most 32 distinct levels and sizes, which require less than one byte to store. A hash map also needs to store a pointer to the head of the collision chain for each hash index. Assuming a load factor of more than 50%, less than two additional pointers are stored per cell. In total, 3 pointers and the key are required per cell, summing up to 25 bytes.

Therefore, the hashed version requires 30% less memory than the linked version.

3.3 Basic Operations

We will now discuss a few basic operations required to work with the octree data structure. Normal lookup as well as more complex operations such as octree construction will be discussed. The special properties of the octree can be employed for optimal performance. The solution for the hashed octree often differs from the common linked variant.

3.3.1 Cell Access

In the simple case in which position and size of the cell is known, the linked version traverses down the tree starting at the root. At level i, the i-th bit of each coordinate of the cell position determines which of the eight children to traverse to. The smaller the cell, the lower its level and the more traversal steps are required to reach it. Because any non-degenerate tree of n cells has $k = O(\log n)$ levels, retrieving a cell takes $O(k)$ operations on average.

If the cells are stored in hash map, the cell key (position and size) is transformed to an index using the hash function. We use a simple sequence of a few shifts, additions and exclusive ors (XORs) to compute a 32 bit value from the 104 bit key. The hash index is the remainder of the division by the size of the hash table.

We use this index to access the head of a list of cells all mapping to this index. On average, this list has $O(\alpha)$ entries, corresponding to the ratio between the total number of cells and hash table size. Every time a filling factor of 80% is reached, the hash table is increased by 50% and the elements are rehashed. Thus, our filling factor α is always between 2/3 and 4/5 and a cell can be accessed in amortized constant time [FT01].

To evaluate a trilinear interpolation at an intermediate position, one needs to find the smallest cell containing this point. The process stays the same for a linked tree: the tree is traversed top down until the smallest cell containing the point of interpolation has been hit. With a hashed version, the smallest cell can be found with a binary search across k levels. As each cell is accessed (or determined to be non-existent) in constant time, the complexity of the binary search is $O(\log k)$. Therefore, cell access is faster for hashed octrees than for the pointered variant if the tree contains enough cells. Some sample performance numbers to support this claim are listed in Table 3.1.

3.3.2 Neighbor Access

Given the position and size of a cell, we can easily compute the corresponding values for its neighbors and perform a normal search in the hash map. For a linked tree, there are special algorithms to optimize traversal to the neighbor cell. The basic concept is to travel upward to the first common super-cell of the two neighbors and then travel downward again using the *opposite* path in one of the coordinates [FP02]. For example, to access the neighbor in the x-coordinate direction, the opposite path accesses the left node whenever the original path accessed the right node at the corresponding level and vice-versa. This opposite path can be computed with a simple XOR operation. Although this optimization does not improve the complexity class of the neighbor search (which is still $O(\log n)$, it does provide a significant speedup.

3.3.3 Node Access

A sample node can only be accessed indirectly through a cell. By access, we also mean testing if a node exists. However, there are eight disjoint hierarchies of cells that link to the same node. It is sufficient to consider the largest cell of each hierarchy. The size of the largest cell can be computed from the node position, by using the fact that cells are always aligned to their size. However, it is unknown which of those cells actually exist. A straightforward approach would check each position sequentially until one cell is found. Instead, we can utilize the structure of the octree to optimize the search in many cases and only check a subset of the positions. The optimization is based on the fact that each cell has seven siblings which belong to the same parent cell. If one sibling exists, so do all others (see Figure 3.3). In the best case, all cells which refer to the node are

part of one common parent. Thus, only one cell has to be referenced. If it does not exist, neither does the node. With or without optimization, the complexity of a node access is equivalent to the complexity of a cell access ($O(k)$ for the pointered octree, $O(\log k)$ for the hashed octree).

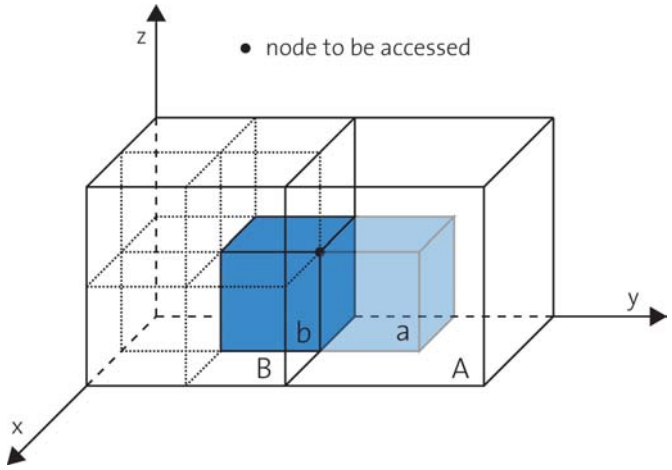


Figure 3.3: Optimized access to corner nodes. Up to eight cells of the same size potentially link to a corner node, but usually only a subset have to be checked. Because the cell (a) does not exist, none of the sibling cells in A do. On the other hand, as cell (b) exists, we know that three other cells in B also reference the same corner node.

3.4 Construction

If the octree is designed to represent an implicit surface, the cells intersecting the surface should be subdivided to capture as much detail of the surface as possible. Each of the children which is again intersected by the isosurface is subdivided as well until the minimum level of subdivision is reached. On the other hand, cells that are not intersected do not affect the surface and should not be subdivided in order to save memory. The construction process needs to determine which cells need to be subdivided based on the scalar function that can be evaluated at discrete locations. Unfortunately, a top-down construction process cannot determine if a surface intersects a cell. Surface features smaller than the current cell size might not enclose any cell corner. Thus, all corner values lie on the same side of the surface and an intersection is not detected (see Figure 3.4). For distance fields, it is at least known that there is no intersection when distance values in each corner is larger than half the cell diagonal. For all other cells and non-distance fields, we have to rely on a look-ahead process, which subdivides each cell for a fixed amount of levels to check for intersection. If no intersection is detected at the small scale, one has to assume that this cell really does not need any subdivision.

If we want to guarantee that no feature larger than the smallest cell is missed, a bottom-up construction must be performed. For this, the function is first evaluated on the entire grid of the smallest scale. We then coalesce all cells which contain nodes on one side of the surface only. Evaluating the distance function on the entire high-resolution grid is expensive and does not match the idea of adaptive sampling.

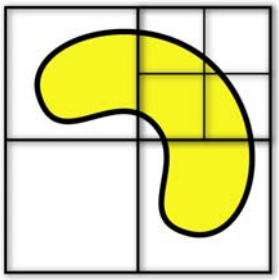


Figure 3.4: Missed features during top-down construction. Small surface features are not detected in a top-down construction process when all corners of a cube lie on the same side of the surface. A look-ahead process is employed to detect features at the subgrid level (top right).

3.4.1 Subdividing a Cell

Subdividing a cell is straightforward. The new cells are inserted into the octree or doubly linked with the parent cell, depending on the version. The new corner nodes on the faces and edges of the parent cell are shared with neighboring cells. The algorithm of Section 3.3.3 is employed to find the references to those nodes. If a node does not exist yet, a new one is added to the array of nodes. Those new nodes are called *hanging* nodes, because they are not surrounded by eight cells. Hanging nodes cause discontinuities of the piece-wise trilinear interpolation across cell borders. To avoid them, the value of a hanging node is not evaluated from the input function, but from the trilinear interpolation of the parent cell. Once a node is completely surrounded by cells, we assign the value from the input function.

3.4.2 Cell and Node Cache

Access to cells and nodes of the octree often occur in temporal clusters. This locality of reference can be exploited to provide faster access to cells and nodes through a small cache. We use a simple directly mapped cache, i.e. a cache entry is evicted as soon as new value uses the same cache line. The hash function can also be used to determine which cache line an entry is mapped to. We simply use another divisor for the modulo operation. In our experiments, the construction of an octree showed around 50% of cache hits for a cache of 100 entries for both nodes and cells.

levels	number of cells	average access level	linked	hashed	speedup	dataset
5	13,897	2	3.42	3.40	-1%	
7	255,753	0.34	20.93	21.54	-3%	
9	4,217,417	0.34	39.92	29.51	26%	
7	89,545	0.30	15.7	17.6	-11%	
9	1,473,373	0.34	32.86	28.9	14%	
10	5,912,649	0.33	46.56	27.26	41%	

Table 3.1: Random access performance comparison. The time (in seconds) to access 50 million cells in random order was measured for the linked and the hashed version of the octree. For octrees with sufficient subdivision levels, the constant access time of the hashed octree can outperform the logarithmic access time of the linked variant.

3.4.3 Blocking

Besides temporal coherence, the traversal algorithms also imply a strong spatial coherence, i.e. there is a high chance that a cell is accessed after its parent cell. Thus, CPU cache lines can be utilized more effectively if small subtrees are stored in a continuous array. Moreover, the tree structure within such a block is implicit and thus, no pointers or extra hash entries need to be stored. For example, eight siblings could be stored in one block and be referenced with one pointer or hash key. On the other hand, the memory for a full subtree is required even if it only contains one single cell. On average, this generates an overhead of one third of the unblocked tree size, because the octree cannot be balanced. Subtree blocking has not been used in our implementation.

3.5 Discussion

An alternative method has been presented to store the structure of an octree which represents an implicit surface. Instead of double-linking parent and child cells, a hash map is employed to map the cell position and size to its data. Similar to sorted and hashed associative containers, the pointered octree provides a sorting of its elements according to the cell subdivision, whereas the hashed octree does not allow an efficient breadth-first traversal of the cells. However, a linear traversal is not intended for this application. Thus, the advantage of hashed associative containers can be employed, i.e. random access in constant instead of logarithmic time. Table 3.1 shows a performance comparison of a random cell access between

the linked and the hashed version of the octree. When the octree is deep enough, the hashed version is significantly faster. Additionally, it has been shown that the hashed version requires less memory.

Octrees have also been implemented on GPUs by dependent texture lookups [LHN05]. However, the large amount of fetches and conditional branching is not well suited for GPUs, which commonly use very wide pipelines (of more than 100 pixels) and have a small texture cache in comparison to CPU caches (in the order of 16kB fully associative cache). However, the prominence of hierarchical data structures in computer graphics seem to make their efficient low-level handling very attractive. Software-manageable caches of highly parallel multimedia processors might provide a good playground to investigate this subject.

Chapter 4

Distance Transform

This chapter presents methods for signed distance transform algorithms, which compute the scalar valued function of the Euclidean distance to a given manifold of co-dimension one. If the manifold is closed and orientable, the distance can be assigned a negative sign on one side of the manifold and a positive sign on the other. Triangle meshes are considered for the representation of a two-dimensional manifold as well as point clouds which sample a manifold at discrete points. The distance function is sampled on a semi-regular Cartesian grid. An acceleration structure for triangle meshes is presented which can speed up distance transforms for adaptive sampling such as the octree structure presented in Chapter 3.

For a dense sampling in a narrow band around the surface, algorithms based on scan conversion have proven to be competitive, especially if graphics hardware is employed to speed up the sampling process per primitive. Here, the distance field is obtained by scan converting a number of polyhedra related to the triangle mesh and by conditionally overwriting the computed distance values.

Optical scanning devices sample the surface of an object. The cloud of points serves as a coarse sampling of the zero-contour of the distance field. In the last section of this chapter, an extrapolation method is presented to turn these samples into a continuous function.

The distance u to a manifold S consisting of several elements S_i can be defined as the point-wise minimum of the individual distance fields u_i.

$$u(\bigcup S_i) = min(u_i) \tag{4.1}$$

For signed distance fields, minimization must be carried out with respect to absolute values. If S is a triangle mesh, the S_i can be chosen to represent the triangle faces. It is also possible to use a disjoint union, in which case the S_i become the triangles (excluding the edges), the edges (excluding the endpoints) and the vertices. Collectively, faces, vertices and edges will be denoted as *primitives*.

In order to sample the distance field on a grid, a brute force algorithm would compute the distance of each grid point to each primitive. The distance at one grid point is chosen to be the distance to the closest primitive, thus resulting in the

shortest distance. If the triangle mesh consists of a large number of triangles, this approach is impractical. For an efficient algorithm, as few distances as possible should be computed to find the smallest one. There are two common approaches to achieve this, which differ in the order of the computation and the data structures used to accelerate the process.

The first approach computes the distance values sample per sample. The primitives of the triangle mesh are stored in spatial data structure, such as the kD-tree presented in Section 4.1, to speed up the search of the closest triangle. When computing the distance field value for a given sample, a primitive can be neglected if it is known that a closer primitive exists. The data structure can be employed to skip a large number of primitives: while searching the closest primitive, an upper limit of the final distance is maintained. At the same time, the data structure allows to state a lower bound of the distance for recursive subsets of all primitives. If the lower bound of a subset is larger than the current upper bound of the final distance, the subset can be excluded from the search. This leads to an algorithm logarithmic in the number of primitives of the input mesh. Methods that compute the distance sample after sample are known as image space methods.

The second approach to compute the distance transform is based on scan conversion. Although scan conversion is usually considered an image space algorithm, it is actually an object space method in this particular setting. In order to achieve linear complexity in the number of grid points, a simple polyhedron is assigned to each primitive. The polyhedron encloses the Voronoi cell of the primitive. Voronoi cells consist of all points that lay closest to its corresponding primitive. Thus, the distance to the primitive only has to be computed for grid points inside its polyhedron. Those sampling points are identified using scan-line conversion. Graphics cards provide fast hardware implementation of two-dimensional scan-line conversion. Even the nonlinear distance to the primitive can be computed directly on hardware using the programmability recently introduced.

An implicit representation can also be computed from a surface that is only provided as a cloud of surface points. These points can be interpreted as samples of the distance field at the zero-contour. The conversion of a point cloud to a distance field can thus be interpreted as a data extrapolation problem. To avoid a trivial solution, gradients of the distance field estimated from the point cloud are usually incorporated into the fitting process. Moving Least Squares (MLS) and Radial Basis Functions (RBF) are standard interpolation approaches that have been applied to implicit surface generation and will be introduced briefly. Unfortunately, both approaches are computationally very expensive or are difficult to implement efficiently. More efficient methods have been presented which fit local surface approximations and blend them using the Partition of Unity approach. These methods will be analyzed and extended with respect to distance fields approximation.

4.1 Closest Point Acceleration Structure

Binary space partition (BSP) trees have proven to be effective for many problems in computational geometry. They provide a very flexible tool for sorting and classification. A BSP tree defines a recursive partition of space into convex subspaces. Each node divides its space by a hyperplane into two subspaces, corresponding to the left and the right child node. Each child node is then recursively partitioned again using another hyperplane. The recursion continues until some application-specific condition is met. Each level of the tree provides a non-overlapping partitioning of increasing granularity. This standard definition will be relaxed later to allow small overlaps between neighboring regions.

The tree hierarchy is used as spatial classification of objects. Each object is associated with the node that provides the tightest bounding volume. Therefore, the space defined by one node can be interpreted as the bounding volume of all objects contained in its subtree. Point objects can always be stored at the leaves of the tree. BSP trees were initially introduced to perform hidden surface removal. The static scene geometry is inserted into the BSP tree and during rendering, the viewpoint is compared with this structure to determine visibility. The traversal of the tree determines which objects in a scene are furthest from the camera.

While general BSP trees allow arbitrary split-planes, kD-trees only use axis aligned split planes. Thus, all subspaces are axis-aligned bounding boxes. This greatly reduces the computational complexity of geometric algorithms involving kD-rees. Furthermore, the memory required to store a node is reduced slightly.

4.1.1 kD-Tree Point Query

kD-trees [Sam90] can successfully accelerate a variety of common algorithms, for example collision detection or ray intersection tests [MB90]. In order to measure the distance to a set of objects, the closest of all objects needs to be found first. Here, the kD-tree can be used to quickly exclude distant objects from the set of closest candidates [Sam90]. In comparison to the octree used to represent semi-regular sampling grids, the kD-tree does not stipulate an equally-sized subdivision and can therefore adapt better to the irregular configuration of scattered objects.

The algorithm traverses the nodes of the tree in a recursive fashion, starting at the root. At each node, the distance to the objects stored at that node is measured. If one of the objects is closer than the ones already tested, the object and its distance is memorized. The algorithm then recursively calls the subtree closer to the query point. The recursion stops at leaf nodes. Once the recursion returns, the distance to the closest object of the entire subtree is known. Depending on that distance, it is often possible to skip recursion for the second subtree. The key observation is that the distance to the bounding box is a lower bound for the distances to all objects it contains. Therefore, a subtree only needs to be traversed if its bounding box is closer than the distance to the closest object already visited.

A priority queue can be employed to concurrently search for the k closest objects. The algorithm is then known as a *k nearest neighbor* search. A few optimizations to this basic approach are possible. Comparing squared distances avoids square root extraction and the lower bound distance can be updated incrementally instead of recomputed from scratch for each node.

The smaller the bounding box of a node, the better the chance that the subtree and all its objects are skipped during nearest neighbor query. However, an object that is intersected by a split plane must be stored at the corresponding node. For objects that are small in comparison to the size of the bounding box, this degrades performance. Therefore, the placing strategy for the split-planes during tree construction has a great impact on the query efficiency. The general goal for a tree of some predefined depth is to store objects in nodes that are as small as possible. However, recursive construction requires an inaccurate a-priori rating of the quality of a split-plane. For example, surface area heuristic (SAH) is a common local greedy approach used in ray-tracing applications [MB90].

Two methods exist to push an object further down the tree into smaller nodes, even if they are intersected by a split-plane. The first one is to simply split the object along the plane. Unfortunately, objects cannot always be split arbitrarily or splitting causes undesirable overhead. For example, cutting a triangle mesh increases the size of the mesh. The second method references the object in both subtrees without splitting it. While this avoids splitting overhead, some bookkeeping has to be employed to avoid evaluating the distance to the same object twice if both subtrees are visited during a query.

Instead, we propose to relax the non-overlapping partitioning by the split-planes and associate each sibling node by its own bounding plane. Thus, two sibling nodes can overlap or even form a gap. A small object that has previously been intersected by the split plane can now be bound by one of the two overlapping nodes. Thus, the object can be associated with a smaller node of the corresponding subtree. On the other hand, a gap can reduce the size of both subtrees.

The node overlap can be represented by two parallel split planes at each node. The algorithms which employ kD-trees only need minor modifications to deal with the additional split plane. For example, the lower bound of a far subtree for the k nearest neighbor search and the ray-casting traversal can be evaluated from the split plane of the far node. On average, the traversal of a tree with overlapping nodes visits fewer nodes and tests fewer objects. However, an additional subtree needs to be traversed sometimes due to the overlap, because the upper bound of the near subtree can be slightly larger than the lower bound of the far tree.

In general, an application dependent trade-off between node overlap and the size of objects stored at the node has to be found. One could also allow gaps or coinciding plane only, for example for exact visibility sorting.

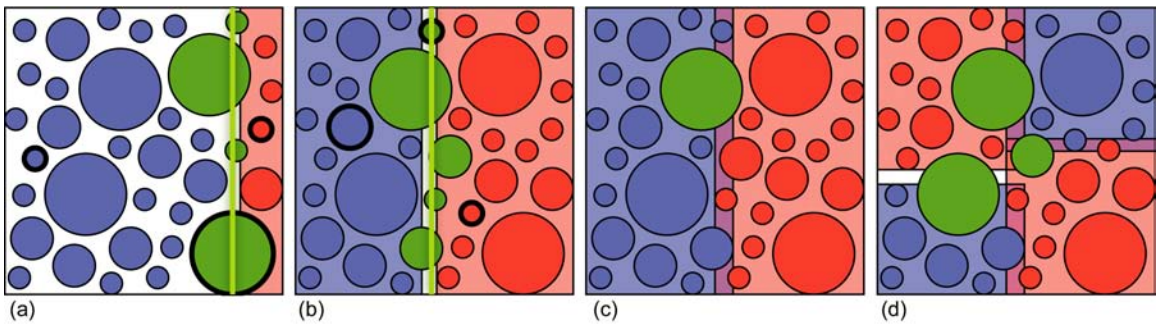


Figure 4.1: kD-tree construction with sibling overlap. An approximate median element is selected from a coarse random sampling (thick circles) to define the split plane (green) which partitions the elements into three sets (blue on the left, green intersecting and red on the right of the split plane). If the partition is uneven, the process is repeated with the larger set (a,b). The elements intersected by the split plane cannot be assigned to one of the two bounding volumes. If two parallel split planes are stored per node, the bounding volumes can overlap or form a gap (c,d). Thus, only big elements (green) are stored at the nodes while smaller ones can be assigned to one of the two child nodes. The process is then repeated to construct the two subtrees (d).

4.1.2 Construction

The construction of a kD-tree with node overlaps will now be discussed in more detail. Several properties have to be optimized concurrently during construction:

- The tree should be well balanced.
- Node overlaps should be minimal.
- Objects should be stored at nodes with a small bounding box.
- Objects stored at the leaves should be evenly distributed.

An optimal solution cannot be found usually, either because of oppositional goals or because the optimization is computationally too involved. Thus, constructing a kD-tree involves a set of heuristics, based on a divide and conquer approach. First, the entire set of objects is divided into three subsets, one to be stored at the root node and one for the left and the right subtree respectively. Each subtree set is then divided again recursively, until the subsets are small enough to be stored in a leaf. The orientation of the split-planes is usually chosen along the axis of current maximum variation or in a round-robin fashion.

The main process of dividing the set into three subsets is based on the median selection algorithm[1]. Ideally, the two subsets corresponding to the subtrees would be of equal size with minimal overlap. In a first step, a median-plane is chosen at the median position of a coarse random sampling of all objects. The objects are then partitioned depending on their position to the left, to the right or on the median-plane (see Figure 4.1). If the partitioning turns out to be uneven, the smaller set already can be assigned to one of the subtrees. The larger set and the middle set are merged and partitioned along a new guess of the median-plane. The working set is reduced in each iteration and the process will eventually converge. In a final step, the objects of the middle set intersecting the median-plane have to be assigned to the node or one of the subtrees.

To achieve this, the objects of the middle set are sorted according to their starting position along the split-coordinate. Each starting position is a candidate for the bounding plane of the right subtree. The best candidate is chosen by a final sweep over the sorted objects. Initially, the large objects of the middle set are assigned to the node, all other objects are assigned to the right subtree. The position of the bounding planes for this configuration has been maintained during the partition process. Then, one object after the other is removed from the right subtree and assigned to the left subtree. For each possible configuration, the position of the two bounding planes and the amount of overlap can be determined. The best of all configurations is then chosen for that node. The process is depicted in Figure 4.1.

The complexity of the partitioning process for one node is linear. The random sampling method during median selection guarantees that only a constant amount of iterations are required on average. Also, the number of objects which intersect the median-plane is independent of the problem size. Therefore, the superlinear complexity of sorting has no impact on the complexity class. Overall, the kD-tree construction has complexity $O(n \log n)$.

4.1.3 Discussion

kD-trees are a versatile data structure that can accelerate many algorithms in computer graphics. In contrast to the octree presented in the last chapter, there is no direct mapping of position to tree nodes and thus, node hashing cannot be employed. However, subtree blocking and node caching are also applicable to kD-trees.

[1] The median selection algorithm divides a set into two subsets of half the size, so that all elements of the first subset are smaller than the ones of the second subset. The very general predicate "*smaller*" needs to define a strict weak ordering [FT01]. However, the position of the objects along the axis of the median-plane only defines a partial ordering: Because objects can overlap, equivalence is not transitive, i.e. x overlaps y and y overlaps z does not imply x overlaps z.

For optimal performance, a kD-tree should store its objects in the leaf nodes. Splitting or multiple references has been employed to achieve this for objects that are intersected by a splitting plane. Instead, a relaxed splitting property has been suggested, which allow sibling nodes to overlap. As a result, objects can always be stored in nodes with a bounding box of similar size. Only small changes are required to adapt kD-tree algorithms to overlapping nodes, but are expected to be more efficient in most cases. The kD-tree construction requires an additional sorting of small subsets but has the same algorithmic complexity as the standard construction process. The relaxed splitting criterion also provides the possibility for gradual updates in dynamic scenes. Instead of re-assigning elements when they move across a split plane, the volume can be dilated so that it still bounds the element. A lazy update procedure can then be employed to rebalance subtrees when their increased overlap has rendered them inefficient.

4.2 Scan Conversion Based Algorithm

Computing the distance field based on a Voronoi diagram is a simple task. For each given sample point, one would first identify the Voronoi cell in which it is contained and then calculate the distance to the associated site. If a full grid of samples is required, one would use an object-space approach, i.e. loop over the cells and identify all points inside one cell. However, the computation of Voronoi diagrams is not easier than the computation of distance fields.

A different approach to compute distance fields was presented in [Mau03]. His Characteristics/Scan-Conversion (CSC) algorithm computes the signed distance field for triangle meshes on a regular grid up to a maximum distance d. The CSC algorithm does not try to find exact Voronoi cells, but instead computes bounding volumes of the Voronoi cells using the connectivity of the triangle mesh. Therefore, only a small part of the distance field has to be calculated for each primitive. The algorithm will be outlined in Section 4.2.2.

The process of identifying all samples contained in the bounding volume is known as scan conversion. In computer graphics, scan conversion is a key operation in the rasterization-based rendering pipeline and is efficiently performed by standard graphics cards. By reading back the frame buffer data, the computing power of graphics cards becomes available for more purposes than just rendering. A hardware implementation of the CSC algorithm is presented in Section 4.2.4. In addition, the method has been revised such that it correctly handles vertices where both convex and concave edges are adjacent.

Although the scan conversion runs faster on the graphics hardware, the overall speed up gained is minimal because of the large amount of geometry that has to be sent to the graphics card. In Section 4.2.3, an optimized type of bounding polyhedra is presented which greatly reduces the geometric complexity of the algorithm.

4.2.1 Generalized Voronoi Diagram

The Voronoi diagram of a finite set of points is a partitioning of space into cells [HKL+99]. Each site (i.e. point) is associated with a cell containing all points for which this site is the nearest one. Points on cell boundaries have more than one nearest site. Voronoi diagrams can be constructed with a graphical method by drawing the graphs of the sites' distance fields. For each single point site, the graph is a vertical circular cone which opens downwards at a fixed angle. By definition, the lower envelope of the graph, i.e. the point-wise minimum, is the Voronoi diagram.

To compute a regular sampling of the Voronoi diagram, the graph of each site is tessellated and rasterized using graphics hardware. The point-wise minimum is naturally determined using the z-test. If the graphs are drawn in distinct colors, the frame buffer contains a sampling of the Voronoi diagram [HKL+99]. The depth buffer contains the distance field.

In three dimensions, the same approach can be used to obtain the distance field and the Voronoi diagram on graphics hardware. However, the rasterization has to be done slice by slice. On a slice, the graphs of the distance field are a hyperboloid of revolution of two sheets. A disadvantage of this method is that it requires accurate rendering of curved surfaces, requiring tessellations in the order of 100 triangles per cone. The 3D version even requires doubly curved surfaces which strongly limits the number of primitives that can be handled.

A straightforward extension is to allow sites to be manifolds instead of just points, leading to generalized Voronoi diagrams (GVD). For this purpose, sites will be restricted to the points, edges and faces of a closed and oriented triangle mesh. For a line segment e.g. the graph is a 'tent' consisting of two rectangles and two half cones. Hoff et al. [HKL+99] presented algorithms to render GVDs in two and three dimensions. Because the GVD algorithm does not restrict the configuration of Voronoi sites, it is more general than needed for signed distance transforms of triangle meshes. Each primitive of the input surface produces a large number of triangles to render. Thus, the method becomes inefficient for large meshes.

4.2.2 Characteristics/Scan-Conversion Algorithm

Ideally, the distance graph would only be computed for all points inside the Voronoi cell of the primitive. However, the geometric complexity of Voronoi cells is related to the complexity of the triangle mesh. Nevertheless, if a point is known to lie outside of a Voronoi cell, the distance to its base primitive does not need to be calculated. This led to the idea of constructing bounding volumes of Voronoi cells. The local configuration of a mesh around a primitive can be employed to construct such a bounding volume. The first such algorithm was presented by Mauch [Mau03].

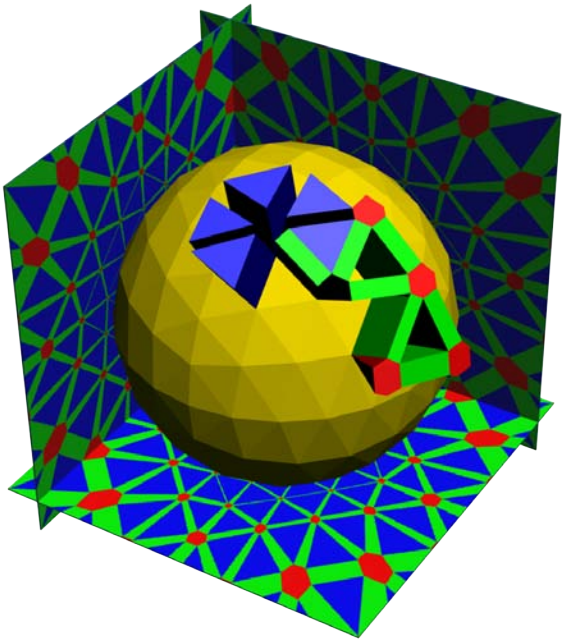


Figure 4.2: Sample Generalized Voronoi cells and slices generated by the hardware-based Characteristics/Scan-Conversion algorithm.

As bounding volumes, he used polyhedra which are possibly larger but of simpler geometric shape than the cells. The distance field can again be calculated by looping over the polyhedra. To correctly treat regions where two or more polyhedra overlap, it is sufficient to take the minimum of all computed values.

The algorithm computes the signed distance field up to a given maximum distance d, i.e. within a band of width $2d$ extending from both sides of the surface. The set of Voronoi sites is chosen as the open faces (triangles), the open edges, and the vertices of the mesh. According to the three types of sites, three types of polyhedra are constructed such that they contain the Voronoi cell as a subset.

Polyhedra for the faces: 3-sided prisms (hereafter called towers) built up orthogonally to the faces (Figure 4.3, light blue).

Polyhedra for the edges: 3-sided prisms (hereafter called wedges) filling the space between towers (Figure 4.3, dark blue). Wedges contain an edge and extend only to one side of the mesh.

Polyhedra for the vertices: n-sided pyramids (hereafter called cones) which fill the gaps left by towers and wedges (Figure 4.3, red). Cones contain the vertex and extend to one side of the mesh.

Both wedges and cones lie on the convex side of the edges and vertices, respectively. The vertices of a closed and oriented triangle mesh can be classified into convex, concave and saddle vertices, depending on their incident edges. If

all of them are convex (concave), the vertex is convex (concave), if both types occur, it is a saddle. Because convex edges become concave and vice-versa when the orientation of the surface is flipped, only convex/concave vertices and saddle vertices are distinguished. If the vertex is a saddle, the polyhedron is no longer a cone and now has a more complex shape. The case of the saddle vertex is not mentioned in [Mau03]. However, a possible solution would be to construct an n-sided pyramid in the same way as for a convex/concave vertex, but on both sides of the surface, and then taking the convex hull of each pyramid.

Besides the topological consistency, a geometric regularity requirement for the mesh has been assumed: At saddle points, all incident faces must keep their orientation when viewed from the normal direction. The normal direction in a vertex is defined simply by averaging all incident face normals. Failure of this assumption would indicate a poor triangulation, which can be fixed by subdividing triangles.

In Figure 4.3, one can see that each polyhedron contains at least the generalized Voronoi cell of its primitive (i.e. face, edge or vertex). Therefore, by scan converting each polyhedron, every voxel lying within the band of width $2d$ will be assigned a distance value. Regions of intersection are scan converted once for each intersecting polyhedron and the minimum value is taken at each voxel.

4.2.3 Prism Scan Conversion Algorithm

Unfortunately, the CSC algorithm produces a large number of bounding polyhedra for complex meshes. Therefore, each polyhedra contains only a few sampling points on average. As a result, the setup-cost of the scan conversion process becomes predominant and makes the method inefficient. For a hardware-based implementation the situation is even worse because the polygons need to be transferred to the graphics card for rasterization.

We propose a modified method that only uses one single bounding polyhedra per triangle, thereby reducing the number of polyhedra to less than a third. Moreover, the average fraction of overlap is reduced. These modified bounding polyhedra result in a significantly speedup over the original algorithm.

Instead of constructing a polyhedra independently per triangle by extrusion, neighboring triangles are incorporated to construct prism-shaped polyhedra. The union of all prisms completely cover the space around the surface, eliminating the need for wedges and pyramids. The price to pay is a slightly more complicated distance field computation: Each polyhedron no longer represents a single site, but seven sites, namely a face, its three boundary edges, and its three vertices. In principle, the minimum of the distances to the seven sites must be calculated. However, this can be done quite efficiently, requiring little more operations than a single distance calculation.

The angle bisector plane between two neighboring triangles forms one of the three lateral boundaries of the new polyhedron. Adding two planes parallel to the face at distances d (the limiting distance used for the distance field computation),

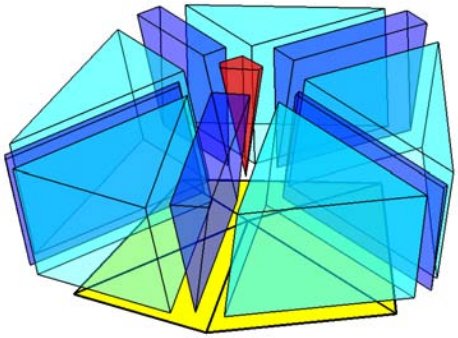


Figure 4.3: Polyhedra constructed on one side of a (yellow) one-ring of the mesh: (cyan) towers, (blue) wedges, and a (red) cone. The polyhedra are moved away from the surface for better visibility.

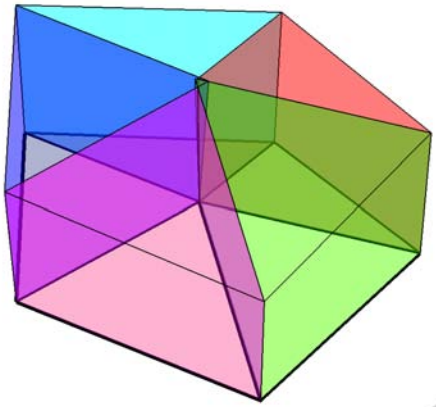


Figure 4.4: Optimized bounding polyhedra for a one-ring of the mesh. Some non-adjacent pairs of polyhedra overlap. The polyhedra extend to the other side of the surface too, which is not shown in this figure.

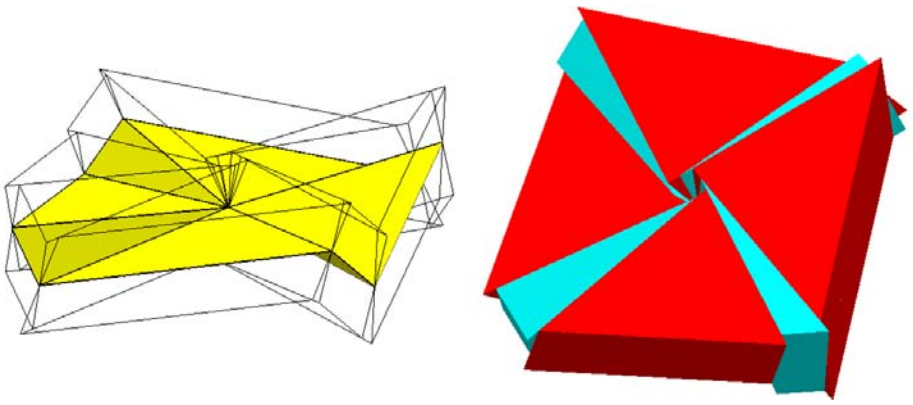


Figure 4.5: Example of a saddle vertex with eight incident triangles. Bounding polyhedra are outlined (left) and filled (right). A gap in the shape of an eight-sided double-pyramid is visible in the center.

a three-sided pyramid frustum is obtained (see Figure 4.4). This bounding polyhedron has the advantage of having a single topological type and only five faces, all of them planar.

While the bounding polyhedra match along the edges of the mesh, this is not true near the mesh vertices in general. Near mesh vertices, the polyhedra can overlap. This is not a problem, it just leads to repeated distance calculations. However, polyhedra can also leave a gap. An example is shown in Figure 4.5. In such cases, the gap must be closed by making some of the polyhedra slightly larger. Appendix A shows that gaps can only occur for saddle vertices.

In order to study the situation near a mesh vertex, a few notations are introduced, see Figure 4.6. Let $\mathbf{c}$ denote the vertex, $\mathbf{x}_0, \cdots, \mathbf{x}_{n-1}$ its neighbor vertices, $F_i =< \mathbf{c}, \mathbf{x}_i, \mathbf{x}_{i+1} >$ the incident faces (all indices are meant modulo n), and P_i the polyhedron constructed for the face F_i. That means that P_{i-1} and P_i are separated by the angle bisector plane of F_{i-1} and F_i which is denoted by A_i. F denotes the union of the F_i and by P the union of the P_i (for $i = 0, \cdots, n-1$).

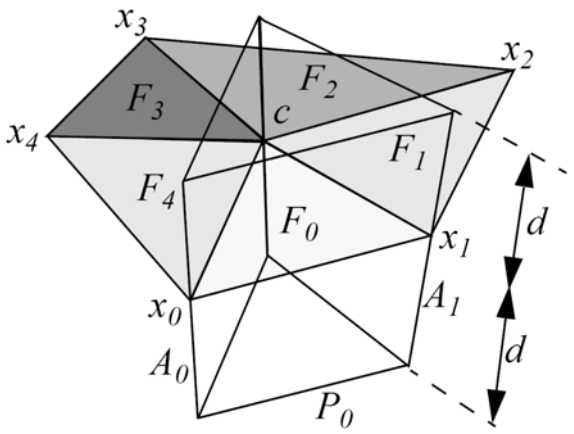


Figure 4.6: A vertex $\mathbf{c}$ with neighbor vertices $\mathbf{x}_i$, faces F_i, angle bisector planes A_i, and polyhedra P_i.

If P completely covers a neighborhood of $\mathbf{c}$, this means that any test point $\mathbf{y}$ near $\mathbf{c}$ is contained in at least one of the P_i. The point $\mathbf{y}$ is contained in P_i if it lies on the right hand side of A_i and on the left hand side of A_{i+1}.

In the reverse case (i.e. left of A_i and right of A_{i+1}), it can also be observed that the antipodal point $2\mathbf{c} - \mathbf{y}$ is contained in P_i. Because in the cycle $A_0, A_1, \cdots, A_n = A_0$ there are as many left-right transitions as right left transition, it follows, perhaps surprisingly, that the covering is point-symmetric w.r.t. the center $\mathbf{c}$. The point symmetry holds for the multiplicity of the covering, not for each single polyhedron.

Therefore, it can be verified that if $\mathbf{y}$ lies neither on the left hand side of all A_i nor on the right hand side of all A_i, it follows that both $\mathbf{y}$ and its antipodal point are covered by P. For the practical test of a complete covering, it is sufficient to use one point on each intersection line $A_i \cap A_{i+1}$. Each test point must lie at least once on the left and on the right side of two other planes. Points lying exactly on a plane should pass the test, too. Also, it has to be noted that, due to point

mesh	vertices	saddles	gaps
sphere6	16386	0	0
torus	3000	1788	0
knot	1440	1378	674
seashell	915	843	148
bunny	34834	30561	516

Table 4.1: Number of vertices, saddle vertices and vertices with incomplete covering by the unmodified bounding polyhedra.

symmetry, full planes can be used for the test, thus there is no need to bother with half-planes.

If the test fails for some of the test points, this means that the corresponding polyhedra must be made larger to avoid a gap. A possible way to do this is to take the centroid of the test points. Polyhedra must be enlarged just as much that they contain this centroid and its antipodal point. In order to ensure that the polyhedra remain pyramid frusta, the modifications have been restricted to parallel shifts of edges.

By looking at a few typical triangle meshes, it can be noticed that there are often more saddle vertices than convex/concave vertices. This can be caused by the geometry itself, but also by the triangulation. Especially, if a quadrilateral mesh is subdivided to a triangle mesh, the diagonals can turn a convex vertex into a saddle vertex. This is why the torus mesh has more than the expected 50% of saddle vertices.

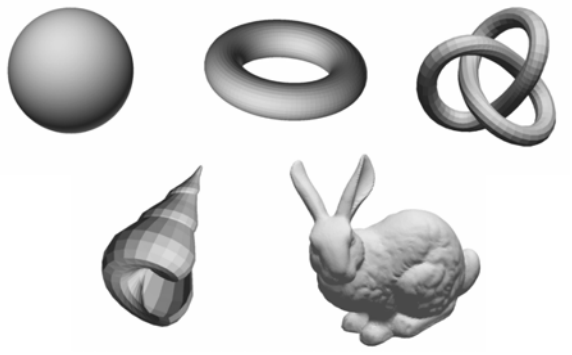


Figure 4.7: Datasets used for experiment. A number of meshes were used to analyze the number of vertices where complete covering fails for the standard prism construction algorithm. The results are listed in Table 4.1.

As mentioned, saddle vertices can lead to gaps between the bounding polyhedra and some extra effort to fill them. However, experiments showed that gaps occur only for some of the saddle vertices. Depending on the mesh characteristics, the percentage of saddle vertices leading to gaps can be quite small (see Table 4.1).

4.2.4 Hardware Accelerated Scan Conversion

The computations for the Characteristics/Scan-Conversion algorithm described in Section 4.2.2 can be divided into two parts. First, all polyhedra are constructed. The rest of the algorithm consists of scan converting these polyhedra and calculating the distances for each grid point inside the polyhedra. For standard mesh and grid resolutions, scan conversion and distance calculation are much more expensive than setting up the polyhedra, especially if the bounding prisms presented in the last section are used.

Therefore, a version that transfers the main workload to the graphics card was implemented. In order to display scenes with a large number of triangles at interactive rates, current generation graphics hardware permit SIMD parallelism for high-speed scan conversion of two-dimensional polygons. The GPU can perform simple operations on a per-fragment basis. To reap the benefits of this computational power, 3D scan conversion was implemented as a series of 2D scan conversions. The programmability of the GPU is crucial for a hardware-based implementation of CSC because the necessary operations exceed standard bilinear interpolation and texture lookup.

The overall slicing process will be described first. Then, an explanation of how each individual slice is rendered will be given.

Slicing Process

The bounding polyhedra are constructed in a setup step. The edges and vertices of the polyhedra are stored in a directed graph. During slice iteration, the graph is traversed along the z-coordinate of the vertices. An active edge table stores all edges intersected by the current slice. When a vertex is passed during slice iteration, all incoming edges of the graph node are deleted from the active edge table and replaced with the outgoing edges.

After sorting all polyhedra according to their first slice intersection, the slice iteration process is initiated. All polyhedra that are intersected by the first slice are copied to an active polyhedra table. Only active polyhedra need to be considered for the current slice.

After rendering all polyhedra intersections, the distance field of the slice stored in the color buffer is read from the graphics card memory. The slice can now be advanced. For all active edges of all active polyhedra, both local and world coordinates of the intersection are incrementally updated. When the advancing slice is moved across a corner of a polyhedron, the active edge table of that polyhedron needs to be updated. All incoming edges of the graph node corresponding to the polyhedron corner are removed and replaced by the outgoing edges, where in and out is defined by the z-direction of the edge. Once the active edge table of a polyhedron is empty, the polyhedron is deleted from the table of active polyhedra. The distance field is computed by repeating these steps until all slices are processed.

Computing one Slice

For each xy-slice of the grid, the cross sections of all polyhedra intersected by that slice are computed. These cross sections are sent to the graphics card, which handles the remainder of the algorithm for that slice. That is, the graphics card computes the distance to the primitive for all points inside its corresponding polygon. The z-test is used for the minimization process in regions where polygons overlap. Similar to the GVD algorithm, the distance graph within one polygon is a doubly curved surface, which need to be approximated by piecewise linear elements. Due to the vast amount of geometry data that would need to be transferred to the graphics card per slice, fine tessellation to approximate the graph has to be avoided.

Instead, one can use the observation that the vector to the closest point on the primitive is indeed a trilinear function within one polyhedron. A fragment program is then employed to calculate the distance to the triangle from a bilinearly interpolated vector on a per-fragment basis. The operations available in a fragment program include normalization of vectors and dot products. Therefore, it is possible to compute the distance value for all grid points inside a polyhedron slice without further tessellation. At each polygon edge, the vector to the closest point on the primitive is passed to the graphics card as a texture coordinate. The graphics card performs a bilinear interpolation of the texture coordinates within the polygon. For each pixel, the GPU computes the dot product of the texture coordinate and its normalized version. The resulting signed distance is written to a floating point color buffer. The absolute distance is used as the z-value for that pixel. If the z-value is smaller than the current value in the z-buffer at the corresponding location, the signed distance value is written to the color buffer.

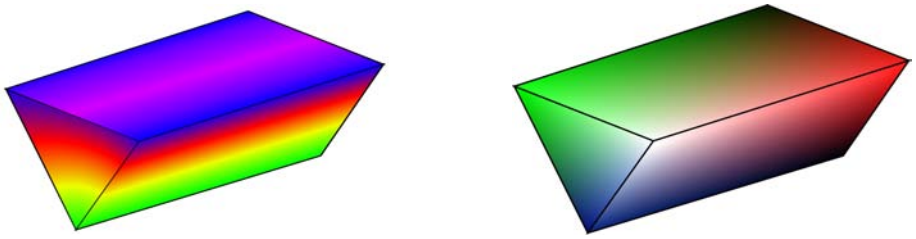


Figure 4.8: Bounding volume of the Voronoi cell of an edge. The distance field within a wedge type polyhedron (left) shows that standard linear interpolation cannot be used draw graphs of the distance field. Instead, the vector to the closest point (right) is interpolated and its length is computed in a fragment program.

The large number of polyhedra which need to be sliced and scan converted significantly contribute to the overall computing time. For each polyhedron slice that has to be processed, rendering calls must be issued to define the geometric shape of the slice. The amount of time required for this operation is independent of the resolution of the slice. If the grid resolution is small in comparison to the triangle size, only a few distance values are calculated per polyhedron slice. Thus,

sending the geometric information to the graphics hardware becomes the bottle neck and parallelism of CPU and GPU cannot be fully exploited. Therefore, the hardware version proves to be faster than the original software algorithm only if the input mesh is relatively small in comparison to the voxel size.

The bounding prism approach greatly reduces the total number of polyhedra. Instead of computing one polyhedron per face, edge and vertex, only one polyhedron per face is computed. Thus, the overall number of polyhedra is reduced to less than one third, thereby reducing the data transfer to the graphics card per slice. The drawback of this approach is that the information about the closest primitive, i.e. face, edge, or vertex, is lost. Consequently, generalized Voronoi diagrams are no longer computed. Additionally, the vector to the closest point on the surface is no longer a trilinear function within the polyhedron. Hence, the approach of interpolating the distance vector within one polyhedron slice and computing the vector length on the GPU fails.

Instead of interpolating the vector to the closest point, the position of the fragment in a local coordinate frame of the triangle is interpolated. The fragment program then searches for the closest corresponding point on the triangle and computes the distance between the two points. The local coordinate frame is defined by three orthogonal axis aligned with the triangle. The r-axis is laid through the longest triangle side such that the three triangle vertices lie on the positive and the negative r-axis and on the positive s-axis orthogonal to it (see Figure 4.9). Their distances from the origin are denoted by a, b and h. The t-axis is parallel to the triangle normal. For each vertex of a polyhedron slice, the texture coordinate is used to define the position of the vertex in this local coordinate system. Texture coordinates are bilinearly interpolated within the polygon and therefore, the texture coordinate of a fragment always holds the position of the fragment in the local coordinate frame.

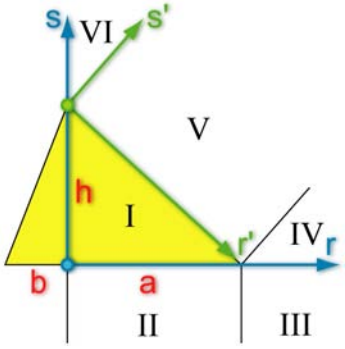


Figure 4.9: Planar distance to a triangle in the plane. To avoid slow conditional branching, the distance to the triangle is computed using two concurrent bases (r,s) and (r',s'). The bases are initialized depending on the sign of the r-coordinate and the vector to the closest point is evaluated depending on regions I-V using conditional writes.

Given these texture coordinates r, s and t, the task of the fragment program is to compute the unsigned distance $D(r,s,t)$ to the triangle. The triangle itself is uniquely described in the same coordinate frame by the three constants a, h and b, as shown in Figure 4.9. These constants are passed to the fragment program as second texture coordinates. This was found to be faster than passing the values as fragment program environment variables, although it involves unnecessary

interpolation of constant values during rasterization.

First, the distance calculation is split into a parallel and an orthogonal part of the triangle plane. Since

$$D(r,s,t) = \sqrt{D(r,s,0)^2 + r^2}, \tag{4.2}$$

the main task is to compute $D(r,s,0)$ which is a two-dimensional problem. The fragment program performs this computation by partitioning the triangle plane into several regions. If r is negative, the coordinate system is reflected at the s-axis. Thus, it is sufficient to treat the six regions labeled I through VI in Figure 4.9. For regions with a positive s-coordinate, the problem is transformed to a second coordinate frame to simplify the location of the closest point on the triangle. Unfortunately, branching is limited in fragment programs. Therefore, the fragment program computes the distance in both coordinate frames and then chooses the appropriate distance depending on the values of r and s. The fragment program in pseudo-code is given in listing 4.1.

```
// Reflect to half-space r>=0 if necessary
if (r<0) { r = -r; a = b; }
// Transform to a 2nd coordinate frame
lenSqr = a^2 + h^2;
r' = (a*r - h*s + h^2) / lenSqr;
s' = (h*r - a*s - a*h) / lenSqr;
// Clamp components of the distance vector
r' = max(-r',r'-1,0); // regions IV, V, VI
s' = max(s',0); // regions I, V
r = max(r-a, 0); // regions II, III
// Compute the distance
if(s<0) // regions II, III
dist = sqrt(r^2 + s^2 + t^2);
else // regions I, IV, V, VI
dist = sqrt( (r'^2 + s'^2) * lenSqr + t^2);
// Place sign
dist = copysign(dist, t);
```

Listing 4.1: Fragment program pseudo-code which computes the distance to a triangle in local coordinates on a per-fragment basis.

4.2.5 Performance Evaluation

As a basis for comparison of performance, the software scan conversion algorithm, which can be downloaded from the author's website [Mau00] was used.

This algorithm was re-implemented such that the scan conversion part was done on the GPU. The performance was measured on a 2.4 GHz Pentium 4 equipped with 1 GB of RAM and an ATI Radeon 9700 PRO graphics card. Only a negligible speedup could be obtained by using this hardware-based variant. In addition, the range of parameters (resolution, width of computational band) where a speedup could be measured, was rather narrow. This performance problem could be tracked down to the overhead caused by rendering too many small polygons.

When using the optimized bounding polyhedra, the speedup delivered on the same machine was significant for a wide range of resolutions and widths. When choosing a band of 10 % of the model extent and a resolution of 256^3 samples, an average speedup close to 5 for the sphere6, knot and bunny models was measured (Table 4.2. For higher resolution as well as for wider bands, the speedup improved (Table 4.5 and 4.4). But also for extremely low sample grid resolutions, the hardware-assisted program performed well. For instance, in the case of a mesh with 131,072 triangles of average area less then 2 on the voxel scale, a speedup of 3.30 was measured. However, it is obvious that the the scan conversion approach, with or without hardware support, is no longer an efficient strategy if sampling density is further decreased. For such problems, an image space method combined with a spatial data hierarchy such as the kD-tree presented in the last section would obviously be more efficient.

The software version from [Mau00] has not been extended to support our optimized bounding polyhedra. However, we assume that one could also measure a performance gain due to the reduced number of prism setup and sample overlap. Nevertheless, the hardware version is expected to achieve a higher speed up when the optimized polyhedra are used, because the geometry needs to be transferred to the graphics hardware slice by slice.

The advantage of the scan conversion approach degrades when the narrow band is large in comparison to the volume the surface encloses. Because the bounding polyhedra for one triangle is computed using only its neighboring triangles, the bounding volumes tend to overlap on the convex side of the surface. The amount of overlap grows superlinearly with the thickness of the narrow band. In order to compute the distance transform in a dense volume around the surface, the fastest solution would be a combination the CSC and the FMM approach. While the CSC algorithm is faster in computing the distance in the narrow band, the FMM algorithm can then be used to compute the distance in regions further away from the surface.

The results prove that current graphics hardware is suitable for supporting the signed distance field computation. A GPU implementation has a larger overhead per polyhedron while sampling the distance field using scan conversion is faster. By reducing the amount of polyhedra to approximately one third, a significant speed up in comparison to the CPU implementation was observed. It was proven that the polyhedra cover the area around triangles, edges and convex or concave vertices up to a user-definable distance. However, the polyhedra can leave a hole in special configurations at saddle vertices. These holes are filled by shifting the

Model	Triangles	Software Algorithm (s)	Hardware Algorithm (s)	Speedup
Sphere6	32,768	6.162	1.346	4.58
Bunny	69,451	19.408	3.737	5.19
Knot	2,880	6.437	1.176	5.47

Table 4.2: Timings (in seconds) for Bunny and Knot data sets. The band width is set to 10% of the model extent, the grid resolution is set to 256^3.

Mesh Size (#triangles)	Software Algorithm (s)	Hardware Algorithm (s)	Speedup
2,048	5.506	0.796	6.92
8,192	5.177	0.918	5.64
32,768	6.162	1.346	4.58
131,072	10.387	3.151	3.30

Table 4.3: Timings (in seconds) with variable input mesh size for 0.1 band width and a 256^3 grid.

Width of Band	Software Algorithm (s)	Hardware Algorithm (s)	Speedup
0.1	6.162	1.346	4.58
0.2	11.701	1.785	6.56
0.4	21.009	2.546	8.25
0.8	34.489	3.724	9.26

Table 4.4: Timings (in seconds) with variable band width for a tessellated sphere with 32,768 triangles 256^3 grid.

Grid Resolution	Software Algorithm (s)	Hardware Algorithm (s)	Speedup
64^3	0.901	0.244	3.69
128^3	1.638	0.482	3.40
256^3	6.162	1.346	4.58
512^3	109.400	6.534	16.7

Table 4.5: Timings (in seconds) with variable grid resolution for a tessellated sphere with 32,768 triangles and 0.1 band width. The speedup factor grows slowly with increasing resolution up to the point where the memory size becomes an issue. In contrast to the software CSC algorithm, the Prism algorithm does not have to keep the full grid in memory.

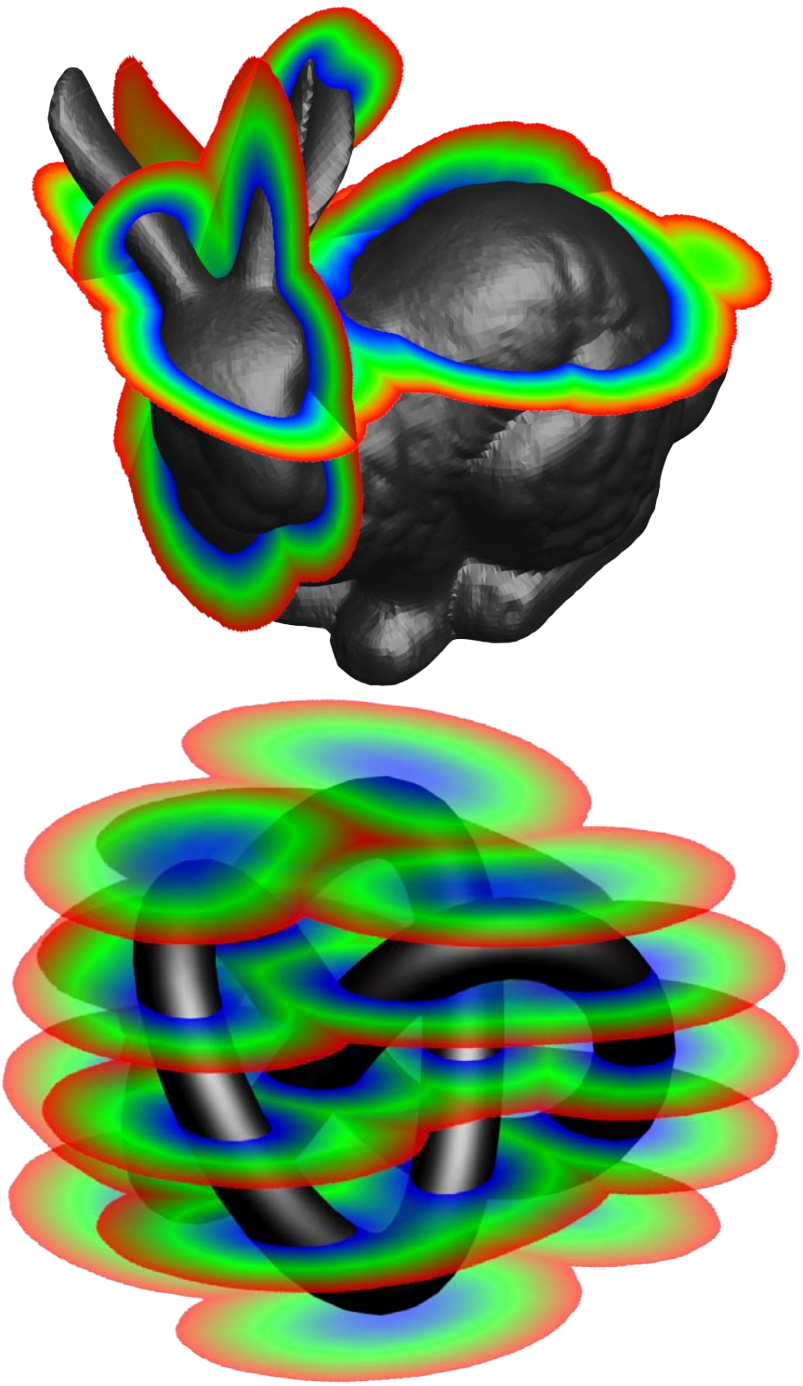


Figure 4.10: Distance transforms of the knot and the bunny model. The distance transform computes an implicit representation of a triangle mesh. The resulting distance field is the scalar valued function of the Eucledian distance to the closest point on the triangle mesh. The image shows several slices of the regularly sampled distance field together with the input model.

sides of the polyhedra outward until they cover the normal of the saddle vertex. This procedure increases the amount of overlap and therefore introduces a certain overhead. However it was shown that gaps do not appear very often for common meshes. For an implementation using graphics hardware, the speed up gained by the reduced amount of geometry outweighs the extra cost of additional distance samples.

4.3 Meshless Distance Transform

All methods presented so far in this chapter relied on a closed triangulated manifold as input of the distance transform. However, surface scanning devices usually produce point clouds without connectivity. Furthermore, the point clouds are often too noisy and incomplete to be triangulated consistently using a Delaunay approach. Instead, implicit representations are much better suited for repairing incomplete data because no topological constraints are required.

The conversion from a set of surface samples to an implicit representation can be stated as a scattered data interpolation problem. Indeed, the point cloud samples the zero-contour of the signed distance field. Thus, a continuous function which interpolates or approximates these zero-distance samples is desirable. Mathematically, the fitting problem [Wen05] can be stated in the following way: From a function $f(\mathbf{x})$, a set of values f_i are given at locations $\mathbf{x}_i$. The goal is to find an approximation $s(\mathbf{x})$ to $f(\mathbf{x})$. Three common data interpolation approaches have been used to gain an implicit definition from point samples: Radial Basis Functions (RBF [CBC+01]), Moving Least Squares (MLS [Lev03]) and Partition of Unity (PU [Nie04]). These approaches will be summarized in the following three sections. Due to the fact that all sample values vanish, the original interpolation problem has to be extended to avoid a trivial solution. Preferably, the solution should approximate a distance field to the surface. The different approaches proposed for each approximation method will be discussed and analyzed in Section 4.3.4. A natural problem of all methods is that although a high-quality distance field can be achieved near the surface samples, the accuracy degrades further away from the surface. Further away from the samples, the distance field can be approximated well by the distance to the closest sample. Thus, a smooth blending between the distance field interpolation near the samples, and the distance to the closest point further away is performed. A scale invariant method to compute the blending weights independent of the sampling density is introduced in Section 4.3.5.

4.3.1 Moving Least Squares

MLS [Lev98] approximates the function values by a polynomial function up to degree n. The polynomial is chosen so that a weighted sum of squared errors are

minimized.

$$\min_{p \in P_n} \sum_i w_i \cdot (p(\mathbf{x}_i) - f_i)^2 \tag{4.3}$$

The polynomial which minimizes the quadratic function can be found by QR factorization of a set of linear equations. The key design choice of the MLS approach is that the error weights w_i depend on the position where the function is evaluated. The weights are a function of the Eucledian distance to the sampling location $\mathbf{x}$: $w_i = \phi(\|\mathbf{x} - \mathbf{x}_i\|)$. Thus, the MLS approximation $s(\mathbf{x})$ can be stated as

$$s(\mathbf{x}) = \{p_\mathbf{x}(\mathbf{x}) | p_\mathbf{x} \in P_n, \sum_i \phi(\|\mathbf{x} - \mathbf{x}_i\|) \cdot (p(\mathbf{x}_i) - f_i)^2 \rightarrow \min\} \tag{4.4}$$

This implies that the quadratic minimization problem to find the best polynomial has to be solved for each evaluation of the MLS approximation. To reduce the problem size, one usually chooses a weighting function with compact support. Thus, most of the terms in the sum of Equation 4.4 drop out.

Some applications require the approximating function to interpolate the sampled values. Interpolation can be achieved with a weighting function which grows to infinity around 0, i.e. $\phi(r) = 1/r$ or $\phi(r) = 1/r^2$.

4.3.2 Radial Basis Function

RBF [CBC+01] is a weighted sum of radial functions around the sample positions. Additionally, the sum is augmented by a polynomial of degree at most k.

$$s(\mathbf{x}) = p(\mathbf{x}) + \sum_i \lambda_i \cdot \phi(\|\mathbf{x} - \mathbf{x}_i\|) \tag{4.5}$$

The polynomial p and the coefficients λ_i are chosen so that the RBF interpolates the function values.

$$s(\mathbf{x}_i) = f_i \tag{4.6}$$

The polynomial is used to guarantee an optimally smooth solution to the interpolation problem [CBC+01]. Additional constraints are required to obtain a fully determined linear system of equations. Thus, orthogonality of polynomial and radial coordinates is imposed.

$$\sum_i \lambda_i p(\mathbf{x}_i) = 0 \tag{4.7}$$

In contrast to the MLS, the solution of this one linear system fully determines the RBF interpolation function across the whole domain of definition. However, the problem size grows linearly with the number of function samples, where as the fit of the MLS approach only considers a constant amount of closest samples.

Large linear systems can be solved efficiently only if they are sparse, corresponding to a radial function ϕ of compact support. Unfortunately, only global functions (such as $\phi(r) = r$, $\phi(r) = r^3$ in $\mathbb{R}^3$ or $\phi(r) = r^2 \log(r)$ in $\mathbb{R}^2$) can guarantee smooth data interpolation. Numerical methods to solve this dense linear system are very involved. One such method is called *fast multipole expansion*. The basic idea is to use a pre-conditioner that approximates clusters of radial bases that are far away by a polynomial. We will not go into further details but refer to [BG97] for a discussion of this method.

4.3.3 Partition of Unity

The partition of unity approach [FN80] is another method of data interpolation. It is a combination between polynomial fitting of MLS and a weighted sum of bases of RBF. A MLS polynomial is fitted at each sampling location to locally approximate the function values. To obtain a global approximation, these polynomials are weighted according to their distance to the point of evaluation. Partition of unity refers to the fact that the weighting is normalized.

$$s(\mathbf{x}) = \sum_i \frac{\phi(\|\mathbf{x} - \mathbf{x}_i\|)}{\sum_i \phi(\|\mathbf{x} - \mathbf{x}_i\|)} p_i(\mathbf{x}) \tag{4.8}$$

An approach similar to the one of MLS to achieve value interpolation is known as *Shepard's* method. The weighting function $\phi(r)$ is chosen to have a singularity around zero. When evaluating Equation 4.8 at a sample location $\mathbf{x}_j$, all weights vanish, except for the one corresponding to p_j, which is one. Thus, if the polynomial fit p_j is interpolating, so is $s(\mathbf{x})$. However, weighting by inverse or squared inverse radius do not work well in practice. Here, the infinite support of the weighting function does not only make a practical implementation difficult, but also has an adverse effect on the shape of the interpolation (see Figure 4.11). The Modified Quadratic Shepard's (MQS [FN80]) multiplies the inverse radius by a hat-function to produce a weighting function of compact support.

$$\phi(r) = \frac{max(r_{cutoff} - r, 0)^2}{r^2} \tag{4.9}$$

In [Nie04], the cutoff radius is chosen per basis to include at least 18 neighboring sampling locations. Unfortunately, a fixed compact support will leave the approximation undefined in areas where all weights vanish. Moreover, the weights are no longer monotone with respect to the distance if a different cutoff radius per basis is used. As an alternative, the cutoff radius has been chosen to include a fixed number of sampling points from the location of evaluation. Therefore, the interpolation is defined anywhere and the monotonicity is reestablished.

4.3.4 Distance Field Approximation

All the approximation methods discussed so far can be used to compute a distance field from a set of surface samples. Because the surface samples all lie on the zero contour of the distance field, a straightforward application of the interpolation methods would return a trivial solution (a function which is zero everywhere). Each method uses its very specific technique to avoid a trivial solution.

The MLS approach uses a projection operator [Lev03] to find the closest point on the surface and then evaluates the distance to this point. The projection operator performs the following procedure: First, a plane is fitted through the sample points in a least squares sense (Eq.4.3) using a normal equation. This plane is used as a parametrization to fit a polynomial through the sampling points (Eq.4.4). The starting point is then projected onto this polynomial. The whole procedure is repeated until the point converges.

In the RBF approach [CBC+01], the most common approach to obtain a non-trivial solution is to generate a set of additional distance samples. For each original surface sample, an approximated surface normal is computed. For example by using the least squares plane fitting approach of MLS. Then, two offset samples in the direction of the normal are generated, at a fixed distance on each side of the surface. The function values are set to the positive and negative offset distance, respectively. These offset points establish a gradient in normal direction at the surface positions. Unfortunately, the dense linear system which is already hard to solve grows by a factor of three during this process.

Partition of unity [Nie04] takes a similar approach. But instead of adding additional samples, the gradient of the local polynomial fit is enforced to coincide with the surface normal. With the weighting function of Equation 4.9 used in [Nie04], the interpolation is first order accurate. Thus, the method is able to match the surface normals, if they are provided by the input data. Interpolation does not even require the solution of a linear system. However, a common undesired effect of higher order interpolating filters is overshoot, which produces unsmooth surfaces. The squared inverse distance weighting cannot even produce a valid surface from the five circular samples shown in Figure 4.11.

To avoid overshoot, a non-interpolating filter has been implemented. A polynomial approximation of the Gaussian with compact support as weighting function has been chosen.

$$\phi(r) = max(1 - \frac{r^2}{r^2_{cutoff}}, 0)^3 \tag{4.10}$$

To achieve interpolation of the surface samples, a sparse linear system must be solved for the constant coefficient of the polynomial fits. Fortunately, sparse linear problems can be solved using iterative or direct methods which are effective even for large systems. First order polynomials have been chosen for the local surface approximation. Their gradient is fixed to the approximated surface normal extracted from the point cloud.

4.3.5 Far Field Approximation

Unfortunately, the methods presented can only provide a good approximation of the distance field in a neighborhood of the surface. In areas further away from the surface samples, the approximation quality decreases. Without monotone weighting functions, spurious zero crossings of the distance field were detected, which produce artificial surfaces not present in the input data.

Obviously, the interpolation methods are lacking distance samples away from the surface to guide the approximation process. However, the distance field away from the surface can be approximated well by the distance to the closest sample. Thus, a smooth blending between the distance field interpolation near the samples, and the distance to the closest point further away is performed. If f_{near} denotes the distance approximation of one of the scattered data interpolation methods and f_{far} is the distance to the closest sample, we seek a blending weight w to define the global function

$$f = w \cdot f_{near} + (1 - w) \cdot f_{far} \tag{4.11}$$

The blending weight should be one near the surface samples and smoothly decay to zero further away. Thus, the blending weight is a function of the distance to the samples, which is denoted d. Note that d is only used to compute the blending weight w and has nothing to do with the signed distance approximation f. The distance d to the samples should be scale-invariant, i.e. it should be measured relative to the sampling density. We will now derive a formula for d that is based on statistical analysis of the distances to a fixed number of closest surface samples. These distances $r_i = \|\mathbf{x} - \mathbf{x_i}\|$ are already computed to evaluate the interpolation function f_{near} according to Equation 4.4, 4.5 or 4.8. The statistical analysis of the vector of distances $\mathbf{r} = \{r_0, r_1, ...\}$ uses the relative variance

$$\overline{\sigma}(\mathbf{r}) = \frac{\sigma(\mathbf{r})}{\mu(\mathbf{r})} = \sqrt{\frac{\mu(\mathbf{r}^2)}{\mu(\mathbf{r})^2} - 1} \tag{4.12}$$

where σ is the variance and μ is the mean value. The relative variance $\overline{\sigma}$ turns out to only depend on the scale-invariant distance d to the sampled surface and is independent of the number of surface samples. For a dense regular sampling on the other hand, the relative variance $\overline{\sigma}$ at scale-invariant distance d can shown to be

$$\overline{\sigma}(d) = \sqrt{1/8} \cdot \frac{\sqrt{9 + 18d^2 - 8\left((1+d^2)^{3/2} - d^3\right)^2}}{(1+d^2)^{3/2} - d^3} \tag{4.13}$$

Using Equation 4.12 and 4.13, the scale-invariant distance d can be computed from the first and second order moments of the distances to a fix number of closest sample points. However, the sampling is generally not regular and only a few closest samples are used for the calculation. Therefore, an approximation for the

inverse function can be used, which can be evaluated efficiently. The following approximate linear correlation is actually very accurate for a dense regular sampling (see Figure 4.12).

$$d \sim \sqrt{\overline{\sigma}(\mathbf{r})^{-1} - \sqrt{8}} \tag{4.14}$$

Eventually, the scale-invariant distance d will only be used to compute a smooth blending weight between two accurate distance functions. We use the polynomial Gauss approximation with a cutoff radius of 5. To summarize, the blending weight is computed from the distance vector $\mathbf{r}$ of the closest surface samples by the following formula

$$w(\mathbf{r}) = max\left(1 - \frac{\mu(\mathbf{r})^2/(\mu(\mathbf{r}^2) - \mu(\mathbf{r})^2) - \sqrt{8}}{25}, 0\right)^3. \tag{4.15}$$

This weight is then used to compute the implicit surface definition from the near and far distance approximation according to Equation 4.11.

4.3.6 Discussion

A method which combines three common data interpolation methods to approximate a distance field from a set of surface samples has been presented. The distance is approximated by a normalized weighted sum of polynomial functions fitted through the surface samples. The weights are a Gaussian-like function of the sample distance with compact support. In comparison to weighting functions based on the inverse distance, this prevents the surface to overshoot between the samples. However, a sparse linear system of equations needs to be solved for the surface to interpolate the input samples. The support radius of the Gaussian-like function is adjusted so that at least a few polynomials have non-zero weights. Thus, the area of definition is not bound by a fixed support radius. Finally, the approximation is smoothly blended with the distance to the closest sample. The distance to the closest sample is a better approximation of the distance field in areas away from the surface.

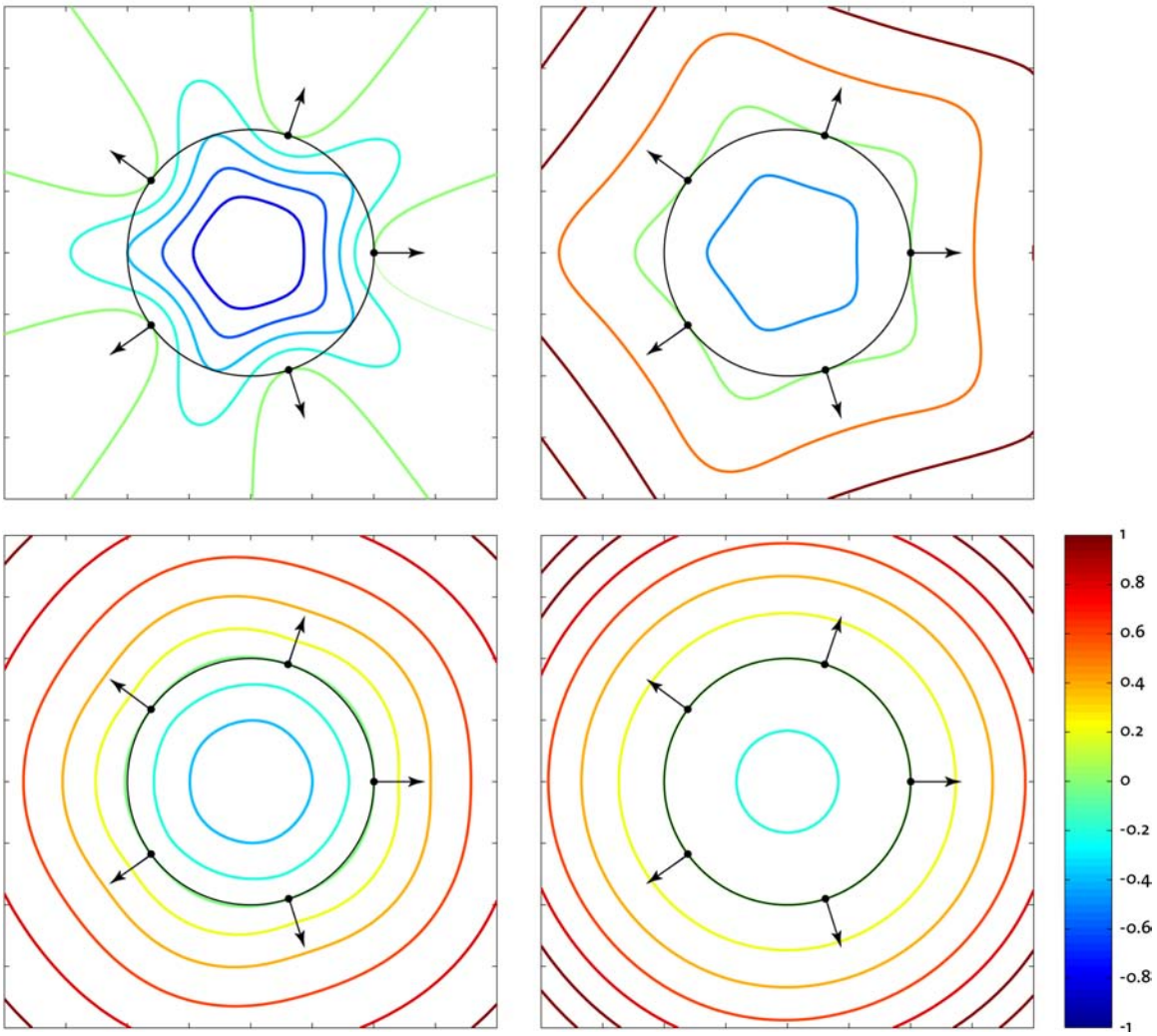


Figure 4.11: Distance field extrapolation from a set of surface samples with normal information. From top left to bottom right: Shepard's method (linear fields blended by partition of unity with inverse quadratic distance weights), extended Shepard's method used in [Nie04] (truncated inverse quadratic distance weights), Radial Basis Functions approach (thin-plate spline), method proposed in this chapter.

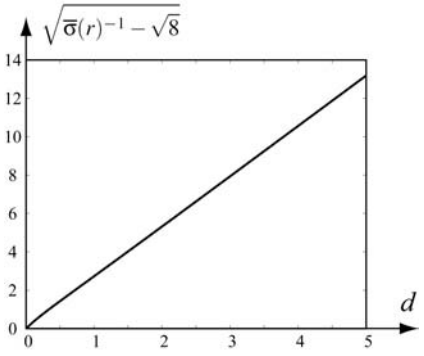


Figure 4.12: Scale invariant distance to samples. The distance to the surface relative to the average sampling density can be approximated from the relative variance $\overline{\sigma}(r)$ of the distances to the closest set of samples. The graph shows the approximation (up to a scaling factor) under the assumption of a dense regular sampling.

Chapter 5

High Order Reconstruction Filters

For implicit surfaces that are represented as a grid of sample values, a continuous definition of the scalar function needs to be retrieved to obtain a closed surface definition. Reconstruction of a continuous function from a regular sampling is performed by convolving the sample pulses with a filter kernel [Hec89]. The choice of the filter kernel is crucial for the quality of the reconstruction.

Current programmable graphics hardware makes it possible to implement general convolution filters in the fragment shader for high-quality texture filtering. However, a major performance bottleneck of higher order filters is the high number of input texture samples that are required, which are usually obtained via repeated nearest neighbor sampling of the input texture [SH05]. This issue will be mitigated in this chapter for filtering with a cubic B-Spline kernel and its first and second derivatives. In order to reduce the number of input samples, cubic filtering is built on linear texture fetches instead, which reduces the number of texture accesses considerably, especially for 2D and 3D filtering. Specifically, a tri-cubic filter with 64 summands can be evaluated using just eight trilinear texture fetches. Often, high-quality derivative reconstruction is required in addition to value reconstruction. The proposed basic filtering method has been extended to reconstruction of continuous first and second order derivatives.

5.1 Higher Order Filtering

OpenGL and DirectX provide two types of texture filtering: nearest neighbor sampling and linear filtering, corresponding to zeroth and first order filter schemes. Both types are natively supported by all GPUs. However, higher order filtering modes often lead to superior image quality. Moreover, higher order schemes are necessary to compute continuous derivatives of texture data.

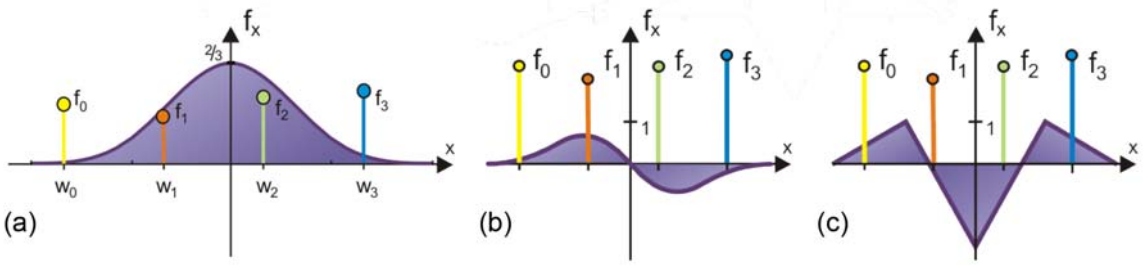


Figure 5.1: The cubic B-Spline and its derivatives. (a) Illustrates convolution of input samples f_i with filter weights $w_i(x)$. (b) and (c) show the first and second derivatives of the cubic B-Spline filter for direct reconstruction of derivatives via convolution.

The implementation of efficient third order texture filtering on current programmable graphics hardware described here primarily considers the one-dimensional case, but extends directly to higher dimensions.

In order to reconstruct a texture with a cubic filter at a texture coordinate x, see Figure 5.1(a), the convolution sum

$$w_0(x) \cdot f_{i-1} + w_1(x) \cdot f_i + w_2(x) \cdot f_{i+1} + w_3(x) \cdot f_{i+2} \tag{5.1}$$

of four weighted neighboring texels f_i has to be evaluated. The weights $w_i(x)$ depend on the filter kernel used. Although there are many types of filters, this chapter will be restricted to B-Spline filters. If the corresponding smoothing of the data is not desired, the method can also be adapted to interpolating filters such as Catmull-Rom Splines.

The filter weights $w_i(x)$ for cubic B-Splines are periodic in the interval $x \in [0,1]$: $w_i(x) = w_i(\alpha)$, where $\alpha = x - \lfloor x \rfloor$ is the fractional part of x. Specifically,

$$\begin{aligned} w_0(\alpha) &= \tfrac{1}{6}(-\alpha^3 + 3\alpha^2 - 3\alpha + 1) \quad & w_1(\alpha) &= \tfrac{1}{6}(3\alpha^3 - 3\alpha^2 + 4) \\ w_2(\alpha) &= \tfrac{1}{6}(-3\alpha^3 + 3\alpha^2 + 3\alpha + 1) \quad & w_3(\alpha) &= \tfrac{1}{6}\alpha^3 \end{aligned} \tag{5.2}$$

Note that cubic B-Spline filtering is a natural extension of standard nearest neighbor sampling and linear filtering, which are zeroth and first degree B-Spline filters. The degree of the filter is directly connected to the smoothness of the filtered data. Smooth data becomes especially important when it is necessary to compute derivatives. For volume rendering, where derivatives are needed for shading, it has become common practice to store pre-computed gradients along with the data. Although this leads to a continuous approximation of first order derivatives, it uses three times more texture memory, which is often constrained in volume rendering applications. Moreover, this approach becomes impractical for second order derivatives because of the large storage overhead. On the other hand, on-the-fly cubic B-Spline filtering yields continuous first and second order derivatives without any storage overhead.

5.2 Fast Recursive Cubic Convolution

This section presents an optimized evaluation of the convolution sum that has been tuned for fundamental performance characteristics of graphics hardware, where linear texture filtering is evaluated using fast special purpose units [SH05]. Hence, a single linear texture fetch is much faster than two nearest-neighbor fetches, although both operations access the same number of texel values. When evaluating the convolution sum, this extra performance should be exploited.

The key idea is to rewrite equation (5.1) as a sum of weighted linear interpolations between every other pair of function samples. These linear interpolations can then be carried out using linear texture filtering, which computes convex combinations denoted in the following as

$$f_x = (1-\alpha)\cdot f_i + \alpha \cdot f_{i+1} \tag{5.3}$$

Where $i = \lfloor x \rfloor$ is the integer part and $\alpha = x - i$ is the fractional part of x. Building on such a convex combination, any linear combination $a \cdot f_i + b \cdot f_{i+1}$ with general a and b can be rewritten as

$$(a+b)\cdot f_{i+b/(a+b)} \tag{5.4}$$

as long as the convex combination property $0 \leq b/(a+b) \leq 1$ is fulfilled. Thus, rather than performing two texture lookups at f_i and f_{i+1} and a linear interpolation, a single lookup at $i+b/(a+b)$ can be multiplied by $(a+b)$.

The combination property is exactly the case when a and b have the same sign. The weights of Equation 5.1 with a cubic B-Spline do meet this property. Therefore the periodic convolution sum can be rewritten as

$$\begin{aligned} w_0(\alpha)\cdot f_{i-1} + w_1(\alpha)\cdot f_i + w_2(\alpha)\cdot f_{i+1} + w_3(\alpha)\cdot f_{i+2} = \\ g_0(\alpha)\cdot f_{x-h_0(\alpha)} + g_1(\alpha)\cdot f_{x+h_1(\alpha)} \end{aligned} \tag{5.5}$$

introducing new functions g_0, g_1 and h_0, h_1 as follows:

$$\begin{aligned} g_0(\alpha) &= w_0(\alpha) + w_1(\alpha) \quad h_0(\alpha) = 1 - \frac{w_1(\alpha)}{w_0(\alpha)+w_1(\alpha)} + \alpha \\ g_1(\alpha) &= w_2(\alpha) + w_3(\alpha) \quad h_1(\alpha) = 1 + \frac{w_3(\alpha)}{w_2(\alpha)+w_3(\alpha)} - \alpha \end{aligned} \tag{5.6}$$

Using this scheme, the 1D convolution sum can be evaluated using two linear texture fetches plus one linear interpolation in the fragment program which is faster than a straightforward implementation using four nearest neighbor fetches. But most importantly, this scheme works especially well in higher dimensions, and for filtering in two and three dimensions the number of texture fetches is reduced considerably, leading to much higher performance.

Some implementation details will now be discussed, including (1) transforming texture coordinates between lookup and color texture and (2) computing the

weighted sum of the texture fetch results. The Cg code of the fragment program for one-dimensional cubic filtering is shown in Listing 5.1. The schematic is shown in Figure 5.2.

As aforementioned, the functions are stored in a lookup texture (called `tex_hg` below) in order to reduce the amount of operations in the fragment program. In practice, a 16-bit texture of 128 samples is sufficient. Note that the functions are periodic in the sample positions of the input texture. Therefore, a linear transformation is applied to the texture coordinate and a texture wrap parameter of `GL_REPEAT` is used for the lookup texture. The linear transformation is incorporated into the fragment program for completeness (see Listing 5.1). However, a separate texture coordinate would normally be computed in the vertex shader.

After fetching the offsets and weights from the lookup texture, the texture coordinate for the two linear texture fetches from the source color texture are computed. Note that the offset needs to be scaled by the inverse of the texture resolution, which is stored in a constant shader parameter.

The rest of the program carries out the two color fetches and computes their weighted sum. Note that B-Splines fulfill the partition of unity $\sum w_i(x) = 1$, and so do the two weights $g_0(x) + g_1(x) = 1$. Therefore, it is not necessary to actually store $g_1(x)$ in addition to $g_0(x)$ in this case, and the final weighting is again a convex combination carried out with a single `lerp()` instruction.

The fragment shader parameters of Listing 5.1 would be initialized as follows for a 1D source texture with 256 texels:

```
e_x = float( 1/256.0f );
size_source = float( 256.0f );
```

The `e_x` parameter corresponds to the size of a single source texel in texture coordinates, which is needed to scale the offsets fetched from the filter texture to match the resolution of the source texture. The `size_source` parameter simply contains the size of the source texture, which is needed to compute filter texture from source texture coordinates so that the size of the entire filter texture corresponds to a single texel of the source texture.

Due to the separability of tensor-product B-Splines, extending this cubic filtering method to higher dimensions is straightforward. Actually, the linear filter optimization works even better in higher dimensions. Using bi- or trilinear texture lookups, four or eight summands can be combined into one weighted convex combination. Therefore, a tri-cubic filter with 64 summands can be evaluated by just eight trilinear texture fetches.

The offset and weight functions for multi-dimensional filtering can be computed independently for each dimension using Equation 4.2. In the implementation, this relates to multiple fetches from the same one-dimensional lookup texture. The final weights and offsets are then computed in the fragment program using

$$g_{\mathbf{i}}(\mathbf{x}) = \prod g_{i_k}(x_k) \qquad \mathbf{h}_{\mathbf{i}}(\mathbf{x}) = \sum \mathbf{e}_k \cdot h_{i_k}(x_k) \tag{5.7}$$

where the index k denotes the axis and $\mathbf{e}_k$ are the basis vectors. Listing 5.2 shows

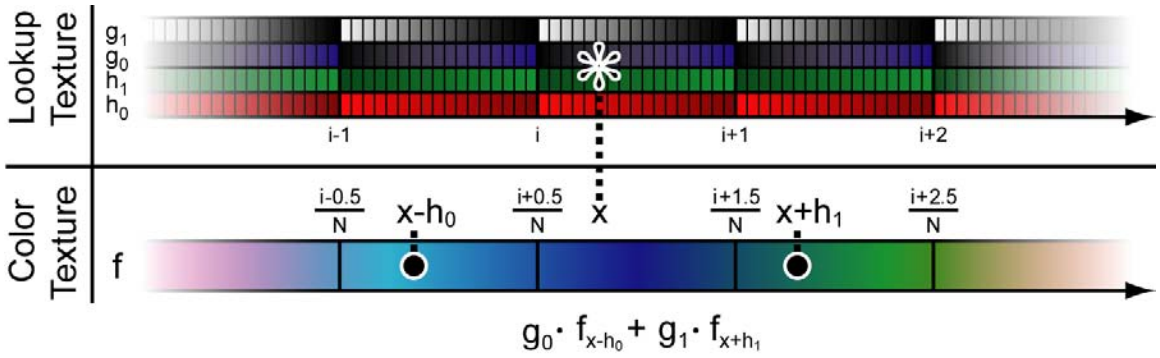


Figure 5.2: Cubic filtering of a one-dimensional texture. In order to reconstruct a color texture of size N, the reconstruction position x is first transformed linearly to the texture coordinate $*$ for reading offsets $h_i(x)$ and weights $g_i(x)$ from a lookup texture. Then, two linear texture fetches of the color texture are carried out at the offset positions (•). Finally, the output color is computed by a linear combination of the fetched colors using the weights $g_i(x)$.

```
float4 bspline_1d_fp(
  float coord_source : TEXCOORD0
  uniform sampler1D tex_source, // source texture
  uniform sampler1D tex_hg,     // filter offsets and weights
  uniform float e_x,            // source texel size
  uniform float size_source     // source texture size
) : COLOR
{
  // calc filter texture coordinates where [0,1] is a
  // single texel (can be done in vertex program instead)
  float coord_hg = coord_source * size_source - 0.5f;

  // fetch offsets and weights from filter texture
  float3 hg_x = tex1D( tex_hg, coord_hg ).xyz;

  // determine linear sampling coordinates
  float coord_source1 = coord_source + hg_x.x * e_x;
  float coord_source0 = coord_source - hg_x.y * e_x;

  // fetch two linearly interpolated inputs
  float4 tex_source0 = tex1D( tex_source, coord_source0 );
  float4 tex_source1 = tex1D( tex_source, coord_source1 );

  // weight linear samples
  return lerp( tex_source0, tex_source1, tex_hg_x.z );
}
```

Listing 5.1: Cubic B-Spline filtering of a one-dimensional texture.

an implementation that minimizes the number of dependent texture reads by computing all texture coordinates at once.

The fragment shader parameters of Listing 5.2 would be initialized as follows for a 2D source texture with 256x128 texels:

```
e_x = float2( 1/256.0f, 0.0f );
e_y = float2( 0.0f, 1/128.0f );
size_source = float2( 256.0f, 128.0f );
```

The special way the source texel size is stored in the `e_x` and `e_y` parameters allows to compute the coordinates of all four source samples with a minimal number of instructions, as shown in Listing 5.2. In three dimensions, the same approach makes it possible to compute all eight source coordinates with only fourteen multiply-add instructions. Filtering in three dimensions is a straightforward extension of Listing 5.2.

5.3 Derivative Reconstruction

In addition to reconstructing the values in a texture map, the reconstruction of its derivatives also has many applications. The gradient g of a scalar field f, in this case a 3D texture, is comprised of its first partial derivatives:

$$\mathbf{g} = \nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)^T \tag{5.8}$$

The most common method for approximating gradient information is to use a simple central differencing scheme. However, for high-quality derivatives we can also use convolution filters and apply the scheme illustrated above for fast evaluation on GPUs. Figure 5.4 shows the quality improvement using higher order gradients for Phong shading of isosurfaces. In order to reconstruct a derivative via filtering, we convolve the original data with the derivative of the filter kernel. Figure 5.1(b) illustrates the first and second derivatives of the cubic B-Spline. Computing the derivative becomes very similar to reconstructing the function value, just by using a different filter kernel.

The fast filtering scheme outlined above can also be applied to derivative reconstruction, by using a derived cubic B-Spline kernel. The convolution weights stated in Equation 5.2 are simply replaced by their derivatives:

$$\begin{aligned} w_0(\alpha) &= \tfrac{1}{2}(-\alpha^2 + 2\alpha - 1) \quad & w_1(\alpha) &= \tfrac{1}{2}(3\alpha^2 - 2\alpha) \\ w_2(\alpha) &= \tfrac{1}{2}(-3\alpha^2 + 2\alpha + 1) \quad & w_3(\alpha) &= \tfrac{1}{2}\alpha^2 \end{aligned} \tag{5.9}$$

In this case the filter kernel weights sum up to zero instead of one: $\sum w_i(x) = 0$. Now, in comparison to Listing 5.1 where the two linear input samples were weighted using a single `lerp()`, the second weight is the negative of the first

```
float4 bspline_2d_fp(
  float2 coord_source : TEXCOORD0,
  uniform sampler2D tex_source, // source texture
  uniform sampler1D tex_hg,     // filter offsets and weights
  uniform float2 e_x,           // texel size in x direction
  uniform float2 e_y,           // texel size in y direction
  uniform float2 size_source    // source texture size
) : COLOR
{
  // calc filter texture coordinates where [0,1] is a
  // single texel (can be done in vertex program instead)
  float2 coord_hg = coord_source * size_source - 0.5f;

  // fetch offsets and weights from filter texture
  float3 hg_x = tex1D( tex_hg, coord_hg.x ).xyz;
  float3 hg_y = tex1D( tex_hg, coord_hg.y ).xyz;

  // determine linear sampling coordinates
  float2 coord_source10 = coord_source + hg_x.x * e_x;
  float2 coord_source00 = coord_source - hg_x.y * e_x;

  float2 coord_source11 = coord_source10 + hg_y.x * e_y;
  float2 coord_source01 = coord_source00 + hg_y.x * e_y;
         coord_source10 = coord_source10 - hg_y.y * e_y;
         coord_source00 = coord_source00 - hg_y.y * e_y;

  // fetch four linearly interpolated inputs
  float4 tex_source00 = tex2D( tex_source, coord_source00 );
  float4 tex_source10 = tex2D( tex_source, coord_source10 );
  float4 tex_source01 = tex2D( tex_source, coord_source01 );
  float4 tex_source11 = tex2D( tex_source, coord_source11 );

  // weight along y direction
  tex_source00 = lerp( tex_source00, tex_source01, hg_y.z );
  tex_source10 = lerp( tex_source10, tex_source11, hg_y.z );

  // weight along x direction
  tex_source00 = lerp( tex_source00, tex_source10, hg_x.z );

  return tex_src00;
}
```

Listing 5.2: Bi-cubic B-Spline filtering of a two-dimensional texture.

Figure 5.3: Texture filter quality comparison using bi-linear (left) and bi-cubic (right) reconstruction filters.

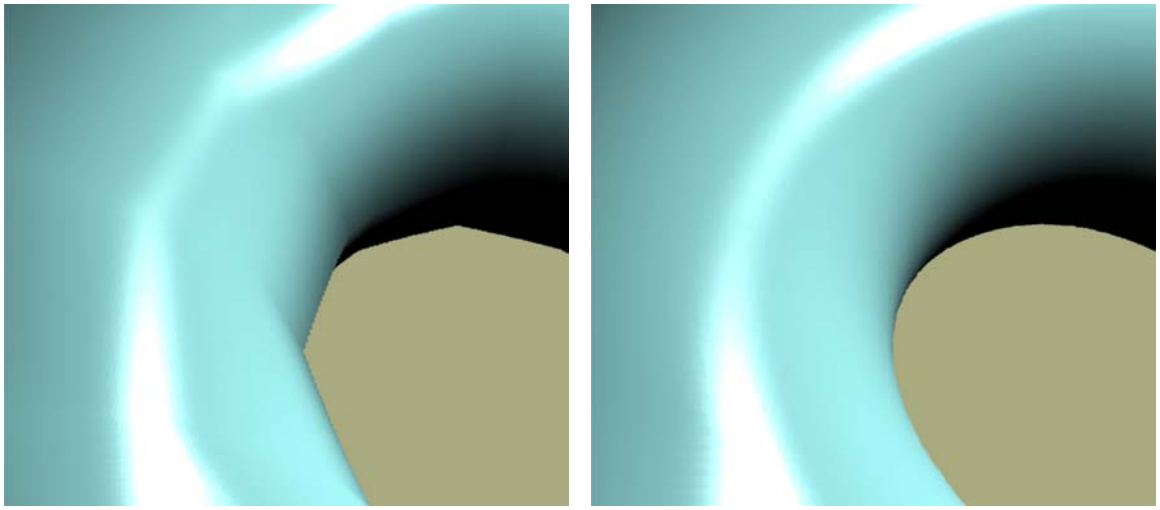


Figure 5.4: Phong shading quality comparison of a torus isosurface using tri-linear (a) and tri-cubic (b) reconstruction filters for both function value and gradient, respectively.

```
float4 bspline_d_1d_fp( ...

  // unchanged from Listing 5.1

  // weight linear samples
  return hg_x.z * ( tex_source0 - tex_source1 );
}
```

Listing 5.3: First-derivative cubic B-Spline filtering of a one-dimensional texture.

one, i.e. $g_1(x) = -g_0(x)$, which can be written as a single subtraction and subsequent multiplication, as shown in Listing 5.3.

In order to compute the gradient in higher dimensions, the corresponding filter kernels are obtained via the tensor product of a 1D derived cubic B-Spline for the axis of derivation, and 1D (non-derived) cubic B-Splines for the other axes. In addition to first partial derivatives, second partial derivatives can also be computed very quickly on GPUs. All these second derivatives taken together comprise the Hessian matrix H, shown here for the 3D case:

$$\mathbf{H} = \nabla g = \nabla^2 f = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial x \partial z} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} & \frac{\partial^2 f}{\partial y \partial z} \\ \frac{\partial^2 f}{\partial z \partial x} & \frac{\partial^2 f}{\partial z \partial y} & \frac{\partial^2 f}{\partial z^2} \end{pmatrix} \tag{5.10}$$

The mixed derivatives in $\mathbf{H}$ (the off-diagonal elements) can be computed using the fast filtering approach for first derivatives which was described above, because the 1D filter kernels are only derived once in this case. For the diagonal elements of $\mathbf{H}$, however, the derivative of the filter kernel itself has to be taken two times. Figure 5.1(c) shows the second derivative of the cubic B-Spline, which is a piecewise linear function. The convolution sum with this filter is very simple to evaluate. Listing 5.4 shows how to do this using three linearly interpolated input samples. In this case, no filter texture is needed due to the simple shape of the filter. The three input samples are simply fetched at unit intervals and weighted with a vector of $(1, -2, 1)$.

For any of the partial derivatives, the weights and offsets are built with the tensor product formula of Equation 5.7. For the axis of derivation, the terms are read from the lookup texture computed with the derived B-Spline filter, whereas for all other axis the texture of the non-derived filter is used. Thus, evaluating all partial derivatives at once requires only one access per dimension into each of the two lookup textures, e.g. a total of six lookups for 3D.

```
float4 bspline_dd_1d_fp(
  float coord_source : TEXCOORD0
  uniform sampler1D tex_source, // source texture
  uniform float e_x,            // source texel size
) : COLOR
{
  // determine additional linear sampling coordinates
  float coord_source1 = coord_source + e_x;
  float coord_source0 = coord_source - e_x;

  // fetch three linearly interpolated inputs
  float4 tex_source0 = tex1D( tex_source, coord_source0 );
  float4 tex_sourcex = tex1D( tex_source, coord_source );
  float4 tex_source1 = tex1D( tex_source, coord_source1 );

  // weight linear samples
  tex_source0 = tex_source0 - 2 * tex_sourcex + tex_source1;

  return tex_source0;
}
```

Listing 5.4: Second-derivative cubic B-Spline filtering of a one-dimensional texture.

5.4 Applications and Discussion

This chapter has presented an efficient method for third order texture filtering with a considerably reduced number of input texture fetches. Building on the assumption that a linear texture fetch is as fast or not much slower than a nearest neighbor texture fetch, filtering with a third order filter kernel such as a cubic B-Spline has been optimized to build on a small number of linear texture fetches. A cubic output sample requires two instead of four input samples, a bi-cubic output sample can be computed from 4 instead of 16 input samples, and tri-cubic filtering is possible with 8 instead of 64 fetches from the input texture. In fact, the corresponding fragment programs are more similar to "hand-coded" linear interpolation than to cubic filtering. Another advantage of this method is that all computations that depend on the filter kernel are pre-computed and stored in small 1D lookup textures. This way, the actual fragment shader can be kept independent from the filter kernel in use [HTHG01]. The extension of higher order filters for mip-mapped textures has been presented in [SH05].

The performance penalty of cubic texture filtering is moderate: bi-cubic B-Spline filtering as shown in Figure 5.3 runs at 36% of the performance of bi-linear filtering.

The approach can also be employed for other filter operations in order to reduce the number of texture lookups and value interpolations as long as the neighboring convolution weights have equal signs. For example, possible applications include discrete cosine transform, non-power-of-two mipmap creation, shadow map filtering, depth of field and other effects that can be implemented as a post-processing image filter. In the next chapter, differential isosurface properties are reconstructed on-the-fly from first and second order partial derivatives of the scalar function. In Chapter 7, the technique is used to implement a fast edge detection filter.

A disadvantage of building on linear input samples is that it may require higher precision of the input texture for high-quality results. On current GPUs, linear interpolation of 8-bit textures is also performed at a similar precision, which is not sufficient for tri-cubic filtering where a single trilinearly interpolated sample contains the contribution of eight input samples. 16-bit textures provide sufficient precision of the underlying linear interpolation. Many current-generation GPUs also support filtering of floating-point textures, which would provide even higher precision.

Chapter 6

Isosurface Ray-Casting

This chapter presents a real-time rendering pipeline for implicit surfaces stored as a regular grid of samples. A ray-casting approach on current graphics hardware is used to perform a direct rendering of the isosurface. A two-level hierarchical representation of the regular grid is employed to allow object-order and image-order empty space skipping and circumvent memory limitations of graphics hardware [HSS+05]. Adaptive sampling and iterative refinement lead to high-quality ray-surface intersections. All shading operations are deferred to image space, making their computational cost independent of the size of the input data. The continuous third-order reconstruction filter presented in the previous chapter allows on-the-fly evaluation of smooth normals and extrinsic curvatures at any point on the surface without interpolating data computed at grid points. With these local shape descriptors, it is possible to perform advanced shading using high-quality lighting and non-photorealistic effects in real-time.

The algorithm is generally independent of specific hardware but support for volumetric textures, render-to-texture and looping in fragment programs is assumed (e.g. ShaderModel 3.0). Several shortcomings in existing GPU isosurface rendering approaches are addressed, particularly the lack or inefficiency of advanced shading, and texture memory usage. Modern GPUs are able to perform standard ray-casting of small regularly sampled data sets [KW03a]. However, advanced shading, e.g. curvature-based transfer functions [HKG00, KWTM03], is still the domain of off-line rendering. The amount of texture memory significantly limits data sizes. This problem is aggravated by the demand of high-quality rendering for voxel data of 16-bit precision or more and lossless compression.

6.1 Pipeline Overview

From a high-level perspective, the rendering pipeline is divided into two major stages. The first stage performs ray-casting through the volume in order to compute the ray-isosurface intersection positions per pixel. Empty space skipping is

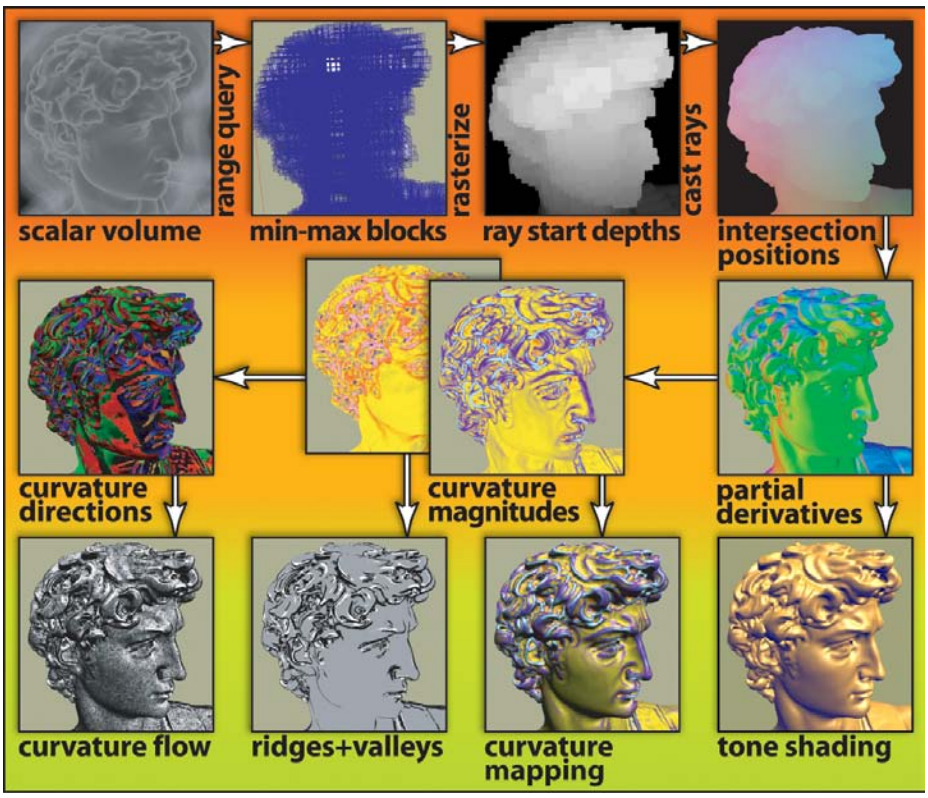


Figure 6.1: Overview of the rendering pipeline. The top row operates with object space complexity until the refinement of ray-isosurface intersection positions. The middle row stages compute differential surface properties with image space complexity, and the bottom row stages perform deferred shading in image space.

employed to perform ray sampling only in areas that contain parts of the isosurface. On-demand caching techniques are employed to dynamically download data to the graphics card that is needed for the sampling process. Because only a small fraction of the grid samples contribute to the definition of the isosurface, this leads to significant reduction of texture memory usage without the need for lossy compression. See Figure 6.3 for an example. Even better memory utilization can be achieved when the grid resolution is adapted to the local amount of detail.

The intersection positions computed in the first stage drive the following stage, which renders the shaded surface using several effects that may also incorporate normal and curvature of the surface. In contrast to direct volume rendering, for isosurfaces only one sample position contributes to the color of a single pixel. Therefore, the method employs a ray-casting pass only for determining ray-surface intersections, and defers the shading computations to image space, where they are evaluated once per visible surface sample only. Thus, the ray-casting stage is the only component of the pipeline that has object space complexity. All other computations have image space complexity [ST90].

As a central component of the rendering pipeline, tri-cubic filtering (Chapter 5)

Figure 6.2: High-quality implicit surfaces are rendered on regular grids, e.g. distance fields or medical CT scans, in real-time without pre-computing additional per-voxel information. Gradients with C^1 continuity, second-order derivatives, and surface curvature are computed exactly for each output pixel using tri-cubic filtering. Applications include surface interrogation and visualizing levelset computations by color mapping curvature measures (top), and ridge and valley lines (bottom left and right).

has been employed throughout. Cubic filters allow for precise evaluations of differential operators of the scalar volume, such as first and second partial derivatives components of the gradient and Hessian matrix. These derivatives are computed at the exact positions of ray-isosurface intersections specified by the intersection image. They are then used to compute differential properties of the isosurface, such as the normal and curvature, which both play a vital role in visualization, modeling and simulation.

The output image is generated with a variety of effects that build on the differential surface properties previously computed. The gradients can be used for all shading models that require a surface normal, such as standard Blinn-Phong [Bli77], tone shading [GGSC98], or high-quality reflections and refractions. Curvature measures can be mapped to colors via 1D or 2D transfer

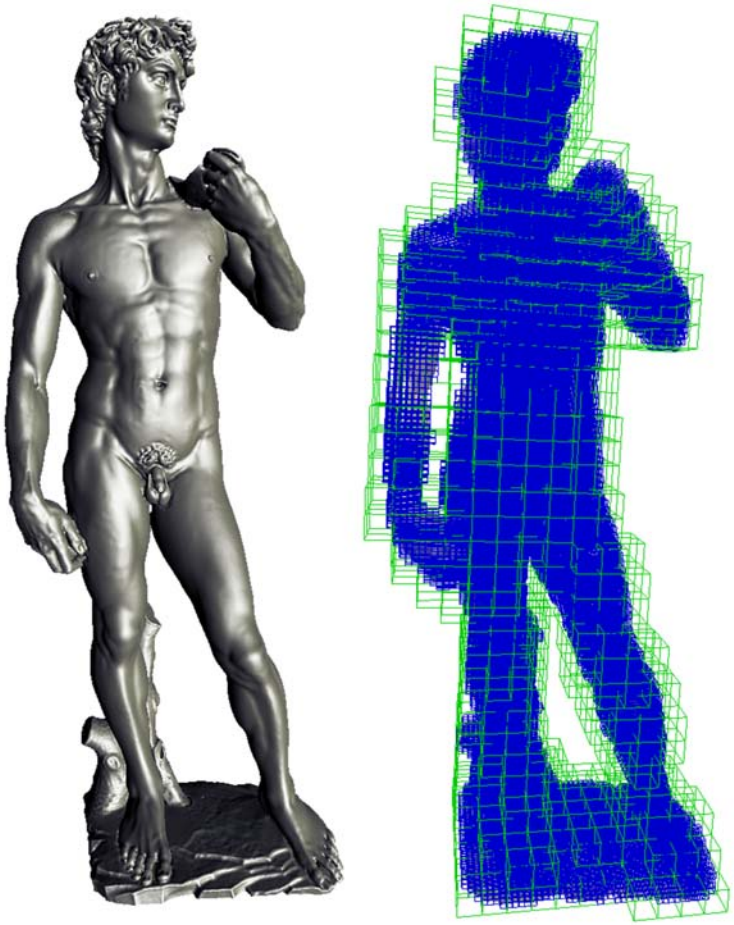


Figure 6.3: Michelangelo's David extracted and shaded with tri-cubic filtering as isosurface of a 576x352x1536 16-bit distance field at 10 fps. The distance field is subdivided into two levels: a fine level for empty space skipping during ray-casting (blue) and a coarse level for texture caching (green).

functions, which are well-suited for shape depiction [KWTM03]. For example, accessibility shading [Mil94], or ridge and valley lines without generating actual line primitives. Pixels that correspond to ridge or valley areas are identified on a per-pixel basis via a curvature transfer function [KWTM03]. Curvature directions are also effective shape cues, and the curvature field on the isosurface is illustrated with image space flow advection [Wij03]. Modeling operations for implicit surfaces, such as Constructive Solid Geometry (CSG) and morphing [COSL98], can be incorporated directly into the rendering pipeline.

In summary, the combination of real-time performance and high-quality yields a general-purpose rendering front-end for many powerful applications of implicit surfaces. The major contribution is a system that integrates the following:

- High-quality shading with non-photorealistic effects using on-the-fly computation of smooth second-order geometric surface properties.

- Object space culling and empty space skipping without any per-sample cost during ray-casting.
- Adaptive grid resolution and a simple 3D brick cache to significantly alleviate GPU memory limitations.
- Precise ray-surface intersections, by combining adaptive resampling and iterative refinement of intersections.

6.2 Hierarchical Representation

Although a fully hierarchical data structure like the octree presented in Chapter 3 can compactly represent a single isosurface, a change of the isovalue to display would require a time-consuming update procedure. Furthermore, octrees implementations on GPUs are relatively inefficient on current architectures due to the lack of point arithmetic. Instead, a relatively simple but effective two-level hierarchy [HBH03] is used to store adaptive sampling grids in texture memory of the GPU.

The volume is subdivided into two regular grid levels: a fine level to facilitate empty space skipping (Section 6.2.1), and a coarse level to circumvent memory limitations of graphics hardware (Section 6.2.2). We call the elements of the fine subdivision level *blocks*, and those of the coarse level *bricks*. The blocks for empty space skipping are chosen smaller to fit the isosurface tightly, and the bricks for memory management are bigger to avoid excessive storage overhead.

6.2.1 Empty Space Skipping

In order to facilitate object-order empty space skipping without per-sample overhead, min-max values of a regular subdivision are maintained, e.g. the volume is subdivided into small blocks of 4^3 or 8^3 voxels. Thus, it can quickly be determined if a block contains part of the isosurface for a given isovalue. If a block is empty, the sampling process can be skipped for the corresponding ray segment. These blocks do not actually re-arrange the volume. For each block, a min-max value is simply stored in an additional structure for culling. Whenever the isovalue changes, blocks are culled against it using their min-max information and a range query [CSS98], which determines their active status. Because no isosurface intersection can occur in ray segments corresponding to empty blocks, those segments do not need to be sampled during ray-casting. Two possible methods for empty block skipping will be presented in section 6.3.1 and 6.3.2.

6.2.2 Brick Caching

The brick subdivision serves to reduce graphics memory consumption in two ways. Firstly, one can use the same argument as for block culling: a brick does not need to be downloaded to the graphics card if it is not intersected by the isosurface. Secondly, the brick with the lowest amount of details can be downsampled to further reduce memory consumption.

For any possible isovalue, many of the blocks do not contain any part of the isosurface. In addition to improving rendering performance by skipping empty blocks, this fact can also be used for significantly reducing the effective memory footprint of relevant parts of the volume. Whenever the isovalue changes, the corresponding range query also determines the active status of bricks of coarser resolution, e.g. 32^3 voxels. The colored squares in Figure 6.5 depict these bricks with a size of $2x2$ blocks per brick for illustration purposes. In contrast to blocks, bricks re-arrange the volume and include neighbor samples to allow filtering without complicated look-ups at the boundaries, i.e. a brick of resolution n^3 is stored with size $(n+1)^3$ [KE02]. This overhead is inversely proportional to the brick size, which is the reason for using two levels of subdivision.

In order to decouple the volume size from restrictions imposed by GPUs on volume resolution (e.g. 512^3 on NVIDIA GeForce 6) and available video memory (e.g. 256MB), ray-casting can be performed directly on a re-arranged brick structure. Similar to the idea of adaptive texture maps [KE02], we maintain an additional low-resolution reference texture (e.g. 16^3 for a 512^3 volume with 32^3 bricks) storing texture coordinate offsets of bricks in a single brick cache texture that is always resident in GPU memory (e.g. a 512x512x256 texture). However, both the reference and the brick cache texture are maintained dynamically and are not generated in a pre-process. Figure 6.4 illustrates the use of the reference and brick cache textures.

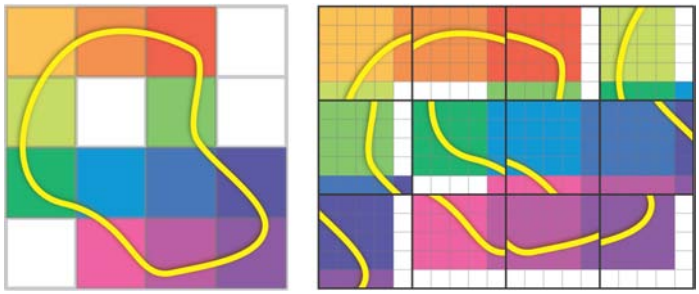


Figure 6.4: A low-resolution brick reference texture (left) stores references from volume coordinates to texture cache bricks (right). The reference texture is sampled in the fragment shader to transform volume coordinates into brick cache texture coordinates. White bricks denote *null* references for bricks that are not resident in the cache.

For a function reconstruction at an arbitrary position, a nearest neighbor lookup is first performed on the reference texture. If the returned value does not indicate a null reference, it is added to the texture coordinate to perform the lookup in the brick cache texture. Unfortunately, this process adds an extra texture indirection to every lookup.

When the isovalue changes, bricks that potentially contain a part of the isosurface are downloaded into the brick cache texture. Inactive bricks are removed with a simple LRU (least recently used) strategy when the storage space which they occupy is required for active bricks. Bricks that are currently not resident in the cache texture are specially marked as null reference at the corresponding position in the reference texture (shown as white squares in Figure 6.4). During ray-casting, samples in such bricks are simply skipped.

6.2.3 Adaptive Brick Resolution

For large data sets, the cache can be too small to fit all bricks that intersect the isosurface. If this happens, selected bricks are downsampled until all bricks fit into the cache texture. Based on an error metric, bricks with small amount of fine surface detail are downsampled first. The error is measured as sum of squared residuals at the sampling positions. At the interface between bricks of different resolution, hanging nodes (see Section 3.4.1 can lead to discontinuities in the reconstructed function. To avoid holes in the isosurface, the hanging nodes of the high-resolution brick are assigned the reconstructed values of the low-resolution brick.

For data sets which require adaptive brick resolution, the reference texture also needs to store the corresponding resolution of each brick. The coordinate of the brick cache texture lookup is then computed as a (per-brick) scale and offset of the volume coordinate.

6.3 Ray-Casting Approach

The basic idea of GPU-based ray-casting is to drive a fragment program that casts rays into the volume. Each pixel corresponds to a single ray $\mathbf{x}(t,x,y) = \mathbf{c} + t\ \mathbf{d}(x,y)$ in volume coordinates. Here, the normalized direction vector $\mathbf{d}(x,y)$ can be computed from the camera position $\mathbf{c}$ and the screen space coordinates (x,y) of the pixel. For each fragment, a ray segment corresponding to the depth range $[t_{in}(x,y), t_{out}(x,y)]$ has to be searched for an isosurface intersection. In order to do this, a fragment program loop is employed to sample the ray segment until a surface intersection is detected. In the simplest case, t_{in} is computed per frame during initialization by rasterizing the front faces of the volume bounding box with the corresponding distance to the camera. Rendering the back faces of the bounding box yields the depths t_{out} of each ray exiting the volume.

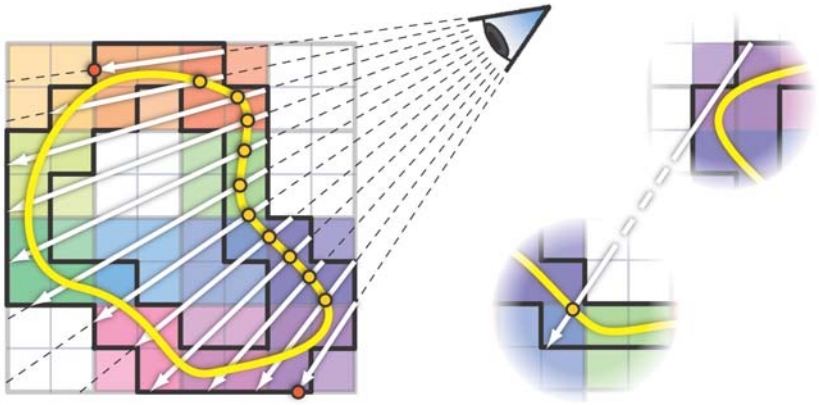


Figure 6.5: Ray-casting with object-order empty space skipping. The bounding geometry (black) between active and inactive blocks that determines start and exit depths for the intersection search along rays (white) encloses the isosurface (yellow). Colored bricks of 2x2 blocks reference bricks in the cache texture (Figure 6.4). White bricks are not in the cache. Actual ray termination points are shown in yellow and red, respectively.

However, this range also includes empty blocks and would therefore not employ empty space skipping. In the following two sections, we are going to present two different strategies for empty block skipping. The first one computes the union of all block segments per pixel during frame initialization. The entire range between the first and the last block is then checked for intersection, even if there are empty segments in between. The second version renders each active block individually. Thus, all empty blocks are indeed skipped. The increased amount of ray segments produce some additional setup cost, but the sample lookup is simplified because segments do not cross brick boundaries.

Once a ray-isosurface intersection has been detected, its position is refined by an iterative bisection procedure, which yields quality identical to much higher constant sampling rates except at silhouette edges (Section 6.3.4). A simple adaptive approach described in Section 6.3.3 improves the quality of silhouette edges.

6.3.1 Bounding Geometry Approach

In order to obtain ray start depths t_{in} for each pixel, the front faces of the block bounding geometry are rendered with their corresponding distance to the camera. The front-most points of ray intersections are retained by enabling a corresponding depth test (e.g. `GL_LESS`). For obtaining ray exit depths t_{out} we rasterize the back faces with an inverted depth test that keeps only the farthest points (e.g. `GL_GREATER`). The geometry of active block bounding faces that are adjacent to inactive blocks is kept in GPU memory for fast rendering.

Because each segment can cross several brick boundaries, each sample per-

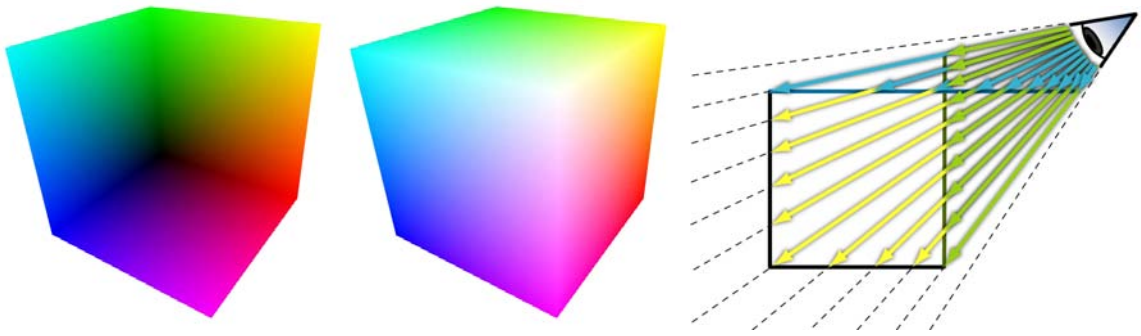


Figure 6.6: Computing block ray segments in a single pass. While only the back faces of the cube are rendered (left), the front face positions can be reconstructed from back face interpolants. The view vector (yellow, right) is interpolated on the back faces of the cube. Additional scale factors are interpolated to reconstruct the positions (blue and green, right) on planes aligned with the front faces. The position furthest from the camera corresponds to the front face.

forms first a lookup in the reference texture to read the offset in the brick cache texture. To avoid this texture indirection, we do not use brick caching when the whole volume fits into the texture memory.

Figure 6.5 shows that this approach does not exclude inactive blocks from the search range if they are enclosed by active blocks with respect to the current viewing direction. Yet, the fragment shader skips to the next sample immediately when a sample is contained in a brick marked as inactive in the brick reference texture (section 6.2.2). Fortunately, most rays hit the isosurface soon after being started and are terminated quickly (yellow points in Figure 6.5, left). Only a small number of rays on the outer side of the isosurface silhouette are traced for a larger distance until they hit the exit position of the block bounding geometry (red points in Figure 6.5, left). The right side of Figure 6.5 illustrates the worst case scenario, where rays are started close to the view point, miss the corresponding part of the isosurface, and sample inactive blocks until they enter another part of the isosurface bounding geometry and are terminated or exit without any intersection.

6.3.2 Block-Level Ray-Casting

In the approach presented in the last section, a single ray segment is sampled per pixel, which crosses both active block and brick boundaries. As a direct implication, empty block skipping is not fully effective and a texture indirection is introduced per sample. To avoid these performance issues, the segments should be sampled on a per-block basis. Unfortunately, the bounding geometry approach cannot be employed because each block would require two rendering passes, one

for the front face and one for the back face. Fortunately, it is possible to compute the front face position of a cube efficiently while rendering the back faces.

The rasterization process performs perspective linear interpolation of vertex values in screen space. For any linear function f in object space, $\frac{f}{w}$ is a linear function in screen space, where w is the homogeneous clip coordinate. Internally, the rasterization process linearly interpolates $\frac{f}{w}$ and $\frac{1}{w}$, from which the function f is reconstructed for every pixel and passed to the fragment shader. Two different homogeneous clip coordinates w_{in} and w_{out} are required concurrently to interpolate the ray segment range $[t_{in}, t_{out}]$, because t_{in} is linear on the front faces while t_{out} is linear on the back faces. Furthermore, the front faces and back faces do not generally coincide in screen space.

With a few tricks, it is still possible to reconstruct the ray segment solely from values interpolated on the back faces of the block. The back face view vector $\mathbf{v} = \mathbf{x} - \mathbf{c}$ points in the same direction as the front face view vector (see Figure 6.6).

$$\mathbf{v}_{out} = f \cdot \mathbf{v}_{in} \tag{6.1}$$

The scale factor $f = \frac{w_{out}}{w_{in}}$ can be interpolated correctly when rasterizing the back face: because $\frac{1}{w_{in}}$ is a linear function in screen space, the scale factor is a linear function on the back face in object coordinates. However, there are up to five back faces and up to three front faces on a cube under perspective projection. As the front and back faces are not aligned, three scale factors f_i are interpolated concurrently across every back face. Each scale factor corresponds to an infinite plane aligned with one of the front faces (Figure 6.6). The ambiguity is resolved in the fragment program on a per-pixel basis: The plane furthest from the camera is the one that corresponds to the front face. In combination with the view vector interpolated on the back face, the front face view vector can thus be reconstructed using

$$\mathbf{v}_{in} = \max(1/f_i) \cdot \mathbf{v}_{out} \tag{6.2}$$

In the vertex program, the scale factor f_i per front face is computed from the view vector $\mathbf{v}$ to the back face in block coordinates. The scale factor at a vertex opposite of the front face can be computed from the view coordinate v_i. Vertices adjacent to the front face always have a corresponding scale factor of 1. Adjacency can be detected with a simple test based on the face normal $\mathbf{n}_i$. The scale factor can thus be calculated using the following formula:

$$f_i = \begin{cases} 1 & : \mathbf{v} \cdot \mathbf{n}_i < 0 \\ \frac{v_i}{v_i - 1} & : \text{otherwise} \end{cases} \tag{6.3}$$

After reconstructing the ray segment, the fragment program needs to sample the corresponding range. For volume rendering, a constant sampling density along the ray is desired to avoid expensive opacity correction [WWH+00]. Thus, the samples would need to be aligned to a global grid. For isosurface rendering however,

the sampling density does not need to be constant. Per ray segment, the range $[t_{in}, t_{out}]$ is sampled at a constant rate that is guaranteed not to fall below a user-specified density.

To employ early ray termination once an isosurface has been hit, the blocks need to be rendered in front to back order. The early z-culling mechanism could then query the depth buffer for an isosurface intersection. If the depth value indicates an intersection in front of the current segment, it does not need to be processed by the fragment shader. Unfortunately, the early-z buffer on current hardware is independent of the full resolution z-buffer and there is no synchronization between the two. The early-z mechanism cannot reflect surfaces rendered with one of the fragment tests enabled. However, the alpha test has to be used to kill fragments of segments that did not intersect the isosurface. Thus, the standard early-z mechanism is not effective in this setting.

However, a manual test can be performed at the very beginning of the fragment program and terminate processing immediately if a segment does not need to be sampled. The segment depth is compared with the current value of the full resolution depth buffer. For this, the depth buffer is concurrently bound as a texture. Texture and frame buffer cache are not synchronized and binding the depth buffer as a texture can lead to inconsistent texture values. However, the early fragment program termination is a performance optimization and as such, a conservative depth test (due to delayed synchronization) does not hamper correctness of the algorithm.

Overall, sampling each ray on a segment-per-block basis introduces some setup overhead. On the other hand, no texture indirection is required to read the cache brick offset and resolution. Empty blocks are fully skipped but early ray termination is not straightforward. Thus, the bounding geometry approach shows better performance when the whole volume fits into the graphics memory and no cache bricks are required. For large data sets that require brick packing and down sampling, block-level ray-casting is the method of choice.

6.3.3 Adaptive Sampling

In order to find the position of the intersection for each ray, the scalar function is reconstructed at discrete sampling positions $\mathbf{p}_i(x,y) = \mathbf{c} + t_i\,\mathbf{d}(x,y)$ for increasing values of t_i in $[t_{in}, t_{out}]$. The intersection is detected when the first sample lies behind the isosurface, e.g. when the sample value is smaller than the isovalue. Note that in general the exact intersection occurs somewhere between two successive samples. Due to this discrete sampling, it is possible that an intersection is missed entirely when the segment between two successive samples crosses the isosurface twice. This is mainly a problem for rays near the silhouette (see Figure 6.7). Guaranteed intersections even for thin sheets are possible if the gradient length is bounded by some value L [KB89]. Note that for distance fields, L is equal to 1. For some sample value f, it is known that the intersection at isovalue ρ

cannot occur for any point closer than $h = |f - \rho|/L$. Yet, h can become arbitrarily small near the isosurface, which would lead to an infinite number of samples for guaranteed intersections.

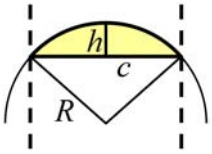


Figure 6.7: Upper error bound of missed features on the silhouette of an object. Given the maximum curvature magnitude κ_1 of the isosurface, the maximal error is given by the thickness h of a circular segment: $error \leq h = R - \sqrt{R^2 - c^2/4}$ where $R = 1/\kappa_1$ is the radius of a circle with curvature magnitude κ_1 and the chord length c is the sampling distance. Thus, it is possible to state the sampling distance required to fulfill a given silhouette error bound.

Adaptive sampling has been used to improve intersection detection. The actual position of an intersection that has been detected is then further refined using the approach described in Section 6.3.4. It has been found that completely adaptive sampling rates are not well suited for implementations on graphics hardware. These architectures use multiple pipelines where small tiles of neighboring pixels are scan-converted in parallel using the same texture cache. With a completely adaptive sampling rate, the sampling positions of neighboring pixels diverge during parallel execution, leading to under-utilization of the cache. Therefore, only two different discrete sampling rates have been used. The *base sampling rate* r_0 is specified directly by the user where 1.0 corresponds to a single voxel. It is the main tradeoff between speed and minimal sheet thickness with guaranteed intersections. In order to improve the quality of silhouettes (see Figure 6.8), a second *maximum sampling rate* r_1 as a constant multiple of r_0 has been used: $r_1 = nr_0$. The implementation currently uses $n = 8$. However, silhouettes are not detected explicitly at this stage, because it would be too costly. Instead, we automatically increase the sampling rate from r_0 to r_1 when the current sample's value is closer to the isovalue ρ by a small threshold δ. In our current implementation, δ is set by the user as a quality parameter, which is especially easy for distance fields where the gradient magnitude is 1.0 everywhere. In this case, a constant δ can be used for all data sets, whereas for CT scans it has to be set according to the data.

6.3.4 Intersection Refinement

Once a ray segment containing an intersection has been detected, the next stage determines an accurate intersection position using an iterative bisection procedure. In one iteration, we first compute an approximate intersection position assuming a linear field within the segment. Given the sample values f at positions $\mathbf{x}$ for the near and far ends of the segment, the new sample position is

$$\mathbf{x}_{new} = (\mathbf{x}_{far} - \mathbf{x}_{near}) \frac{\rho - f_{near}}{f_{far} - f_{near}} + \mathbf{x}_{near} \tag{6.4}$$

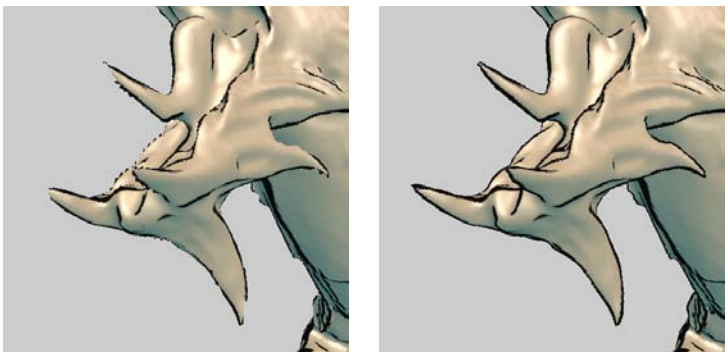


Figure 6.8: The left image illustrates a small detail of the asian dragon model with a sampling rate of 0.5. On the right, adaptive sampling increases the sampling rate to 4.0 close to the isosurface. Note that with the exception of the silhouettes, there is no visible difference due to iterative refinement of intersections.

Then the value f_{new} is fetched at this point and compared to the isovalue ρ. Depending on the result, the ray segment is updated with either the front or the back sub-segment. If the new point lies in front of the isosurface (e.g. $f_{new} > \rho$), $\mathbf{x}_{near}$ is set to $\mathbf{x}_{new}$, otherwise $\mathbf{x}_{far}$ is set to $\mathbf{x}_{new}$ and repeated. We have found empirically that a fixed number of four iteration steps is enough for high-quality intersection positions.

6.4 Deferred Shading

While the last section explained how to compute accurate ray-surface intersections for each pixel, this section describes how to turn the position image into a high-quality rendering using deferred shading. All algorithms described in this section have image space complexity, which means that they are independent of the size of the grid data.

6.4.1 Differential Surface Properties

Chapter 5 described how to quickly evaluate cubic reconstruction filters and their partial derivatives. This section shows that these basic capabilities can be exploited to calculate differential properties of isosurfaces from the scalar volume.

Partial derivatives. The first differential property of the scalar volume that needs to be reconstructed is its gradient $\mathbf{g} = \nabla f$, which can be used as implicit surface normal and for curvature computations. The surface normal is the normalized gradient of the volume, or its negative, depending on the notion of being inside/outside the object: $\mathbf{n} = \pm\mathbf{g}/|\mathbf{g}|$. A tri-cubic B-spline convolution sum is evaluated to compute each of the partial derivatives via eight texture fetches from the 3D volume texture. The Hessian $\mathbf{H} = \nabla\mathbf{g}$ is comprised of all second partial

Figure 6.9: Color mapping of maximum principal curvature magnitude using a 1D color look-up table (dragon data set with 512x512x256 samples).

derivatives of the volume. For mixed derivatives (e.g. $\partial^2/\partial x \partial y$), eight 3D texture fetches are used. For the three diagonal elements of $\mathbf{H}$ (e.g. $\partial^2/\partial x^2$), twelve 3D fetches are necessary. The offsets and weights for the fast convolution described in Chapter 5 can be fetched once for all derivatives with six lookups into a 1D texture. In total, computing all first partial derivatives amounts to 30 texture fetches. Another 60 texture fetches are required to additionally compute all second order partial derivatives. A total of 90 texture fetches seems to be quite excessive. Fortunately, the texture cache is fully effective and the ratio between texture fetches and program instructions is still below the number most current hardwares have been designed for (usually 2 program instructions per texture fetch for both NVIDIA and ATI, except for the most current ATI chip (Radeon X1900) which uses a ratio of 6).

Extrinsic curvature. The first and second principal curvature magnitudes (κ_1, κ_2) of the isosurface can be estimated directly from the gradient $\mathbf{g}$ and the Hessian $\mathbf{H}$ [KWTM03], whereby tri-cubic filtering in general yields high-quality results. The principal curvature magnitudes amount to two eigenvalues of the shape operator $\mathbf{S}$, defined as the tangent space projection of the normalized Hessian:

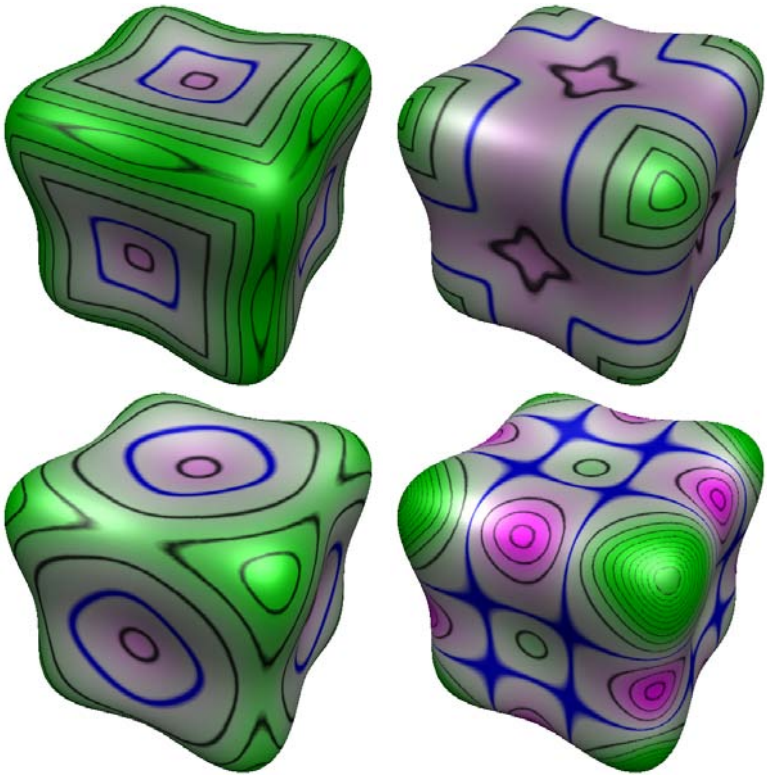


Figure 6.10: Curvature Mapping of a 64^3 synthetic data set. Principle curvatures (top), mean curvature $(\kappa_1 + \kappa_2)/2$ (bottom left), and Gaussian curvature $\kappa_1\kappa_2$ (bottom right). Our renderer is capable of reproducing images from [KWTM03] at interactive rates.

$$\mathbf{S} = \mathbf{P}^T \frac{\mathbf{H}}{|\mathbf{g}|}\mathbf{P}, \qquad \mathbf{P} = \mathbf{I} - \frac{\mathbf{g}\mathbf{g}^T}{|\mathbf{g}|^2} \tag{6.5}$$

where $\mathbf{I}$ denotes the identity matrix. The eigenvalue corresponding to the eigenvector $\mathbf{g}$ vanishes, and the other two eigenvalues are the principal curvature magnitudes. As one eigenvector is known, it is possible to solve for the remaining two eigenvectors in the two-dimensional tangent space without ever computing $\mathbf{S}$ explicitly. This results in a reduced amount of operations and improved accuracy compared to the approach given in [KWTM03]. The transformation of the shape operator $\mathbf{S}$ to some orthogonal basis $(\mathbf{u}, \mathbf{v})$ of the tangent space is given by

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = (\mathbf{u}, \mathbf{v})^T \frac{\mathbf{H}}{|\mathbf{g}|}(\mathbf{u}, \mathbf{v}) \tag{6.6}$$

Eigenvalues of $\mathbf{A}$ can now be computed using the direct formulas for 2x2 matrices. The two eigenvectors of the shape operator $\mathbf{S}$ corresponding to the principal curvature directions are computed by transforming the eigenvectors of $\mathbf{A}$ back to three-dimensional object space.

$$\kappa_{1,2} = \frac{1}{2}\left(\text{trace}(\mathbf{A}) \pm \sqrt{\text{trace}(\mathbf{A})^2 - 4\det(\mathbf{A})}\right) \tag{6.7}$$

$$\mathbf{e}_i = a_{12}\mathbf{u} + (\kappa_i - a_{11})\mathbf{v} \tag{6.8}$$

This amounts to a moderate number of vector and matrix multiplications, and solving a quadratic polynomial.

6.4.2 Shading Effects

After the computation of differential surface properties, the results can be used for deferred shading in image space. Hence, all shading is decoupled from the volume and only calculated for actually visible pixels. This section outlines some of the example shading modes that have been implemented.

Shading from gradient image. The simplest shading equations depend on the normal vector of the isosurface. We have implemented standard Blinn-Phong shading [Bli77] and tone shading [GGSC98] (Figure 6.14), as well as reflection and refraction mapping (Figure 6.13) that index an environment map with vectors computed from the view vector and the normal vector.

Solid texturing. The position image that contains ray-surface intersection positions can be used for application of a solid texture onto the isosurface. The object is parametrized by specifying an affine transformation from object space to texture space coordinates. We also use tri-cubic B-spline filtering instead of trilinear interpolation for improved texture quality (a comparison is depicted in Figure 6.15).

Curvature color mapping. The extrinsic curvature can be visualized on the isosurface by mapping curvature measures to colors via lookup textures. First and second principal curvatures, mean curvature $(\kappa_1 + \kappa_2)/2$ and Gaussian curvature $\kappa_1\kappa_2$ can be visualized using a 1D lookup texture (see Figures 6.9 and 6.10) and provide a good understanding of the local shape of the isosurface. Using a two-dimensional lookup texture for the (κ_1, κ_2) domain allows to highlight different structures on the surface. Figure 6.11 shows approximated accessibility shading [Mil94]. In this case, a simple 1D curvature transfer function has been used to darken areas with large negative maximum curvature. Alternatively, a 2D curvature function could also be employed for this purpose, which would provide finer control over the appearance.

Curvature-aligned flow advection. Direct mappings of principal curvature direction vectors to RGB colors are hard to interpret, see Figure 6.1 (curvature directions). Instead of showing curvature directions directly, they have been visualized with an approach based on image-based flow visualization [Wij03]. In particular, we are advecting flow on the surface entirely in image space [LJH03] by computing advection on a per-pixel basis according to the underlying vector field of principal curvature directions. Image-based flow advection methods can be used

Figure 6.11: Asian dragon data set (512x256x256). Left: tone shading. Right: tone shading blended with accessibility shading, thereby allowing better depiction of local surface details.

Figure 6.12: Dense flow advected in the direction of maximum principal curvature (head of the David data set with 512^3 samples).

Figure 6.13: High-quality surface reflection mapping. The reflection vector is used to perform a lookup into an environment cube map. Cubic gradient reconstruction filter provides smooth normals for alias-free reflection mapping.

on surfaces without parametrization by projecting a 3D flow field to the 2D image plane and advecting entirely in the image [Wij03]. This can be done by simply projecting each 3D curvature direction vector stored in the corresponding floating point image to the image plane immediately before performing advection for a given pixel. Image-based flow advection easily attains real-time rates, which complements the capability of our pipeline to generate the underlying, potentially unsteady, flow field in real-time (see Figure 6.12). A problem with advecting flow along curvature directions is that their orientation is not uniquely defined and thus seams in the flow cannot be entirely avoided [Wij03]. Although these seams are visible when looking closely, they have not been found to be very disturbing in practice. Even though the flow field we are computing from curvature directions contains clearly visible patches (Figure 6.1: curvature directions), the resulting flow has much higher quality (Figure 6.12).

Non-photorealistic effects. Curvature information can be used for a variety of non-photorealistic rendering modes. We have implemented silhouette outlining, taking curvature into account, in order to control thickness, and depicting ridge and valley lines specified via colors in the (κ_1, κ_2) domain [KWTM03]. See Figures 6.16 and 6.10. In our pipeline, rendering modes such as these are simple operations that can be carried out in the final shading pass, usually in combination with other parts of a larger shading equation, e.g. tone shading or solid texturing. The combination of curvature magnitude color maps and curvature-directed flow proves to be especially powerful for visualizing surface shape, e.g. as guidance during modeling.

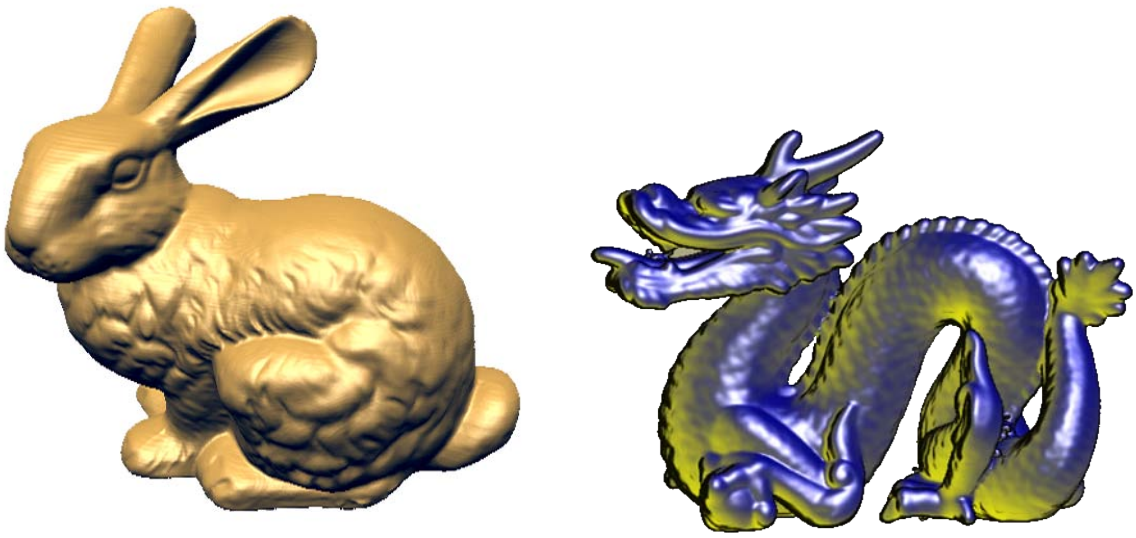


Figure 6.14: A wide range of rendering styles are possible in a deferred shading pipeline. Per pixel phong shading (left) of a CT scan of the bunny model as well as non-photorealistic rendering (right) including tone shading and silhouette outlining of the dragon model.

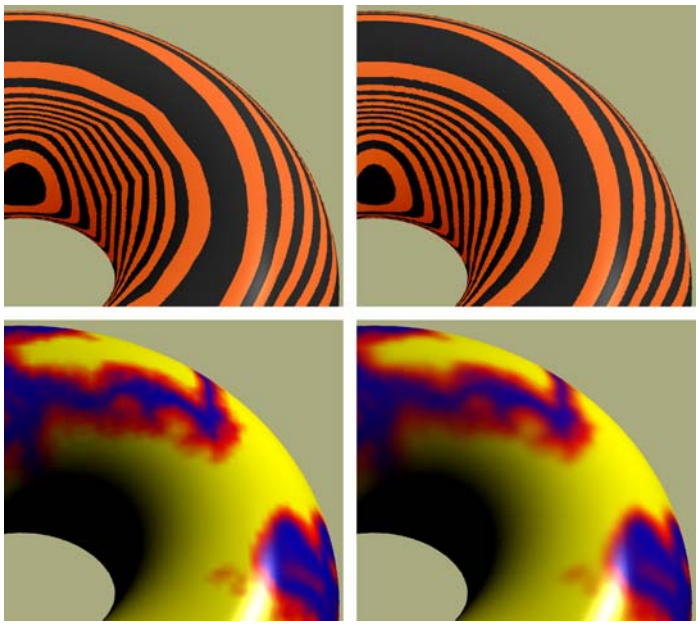


Figure 6.15: Comparison between linear (left) and cubic (right) reconstruction filters. The top row shows reflection lines produced by reflection mapping in a synthetic environment with horizontal stripes. The bottom row shows solid texturing. The superior quality of the cubic reconstruction filter produces uniformer reflection lines and smoother texture mapping.

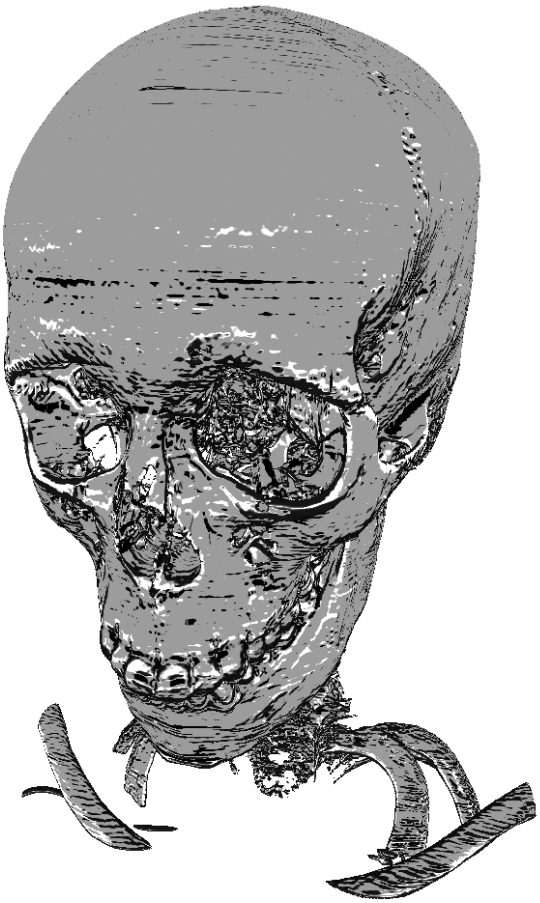


Figure 6.16: Contours modulated with curvature in view direction, and ridges and valleys on an isosurface of a 512x512x333 CT scan of a human head.

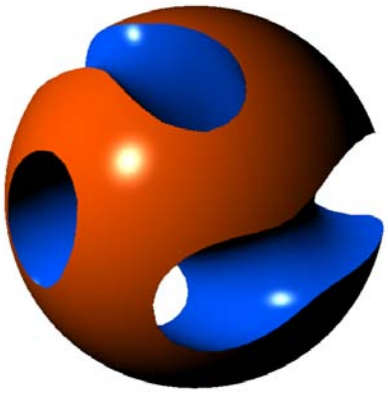


Figure 6.17: Real-time evaluation of CSG operators to cut a knot out of a sphere. The implicit definition of the CSG model is computed from the implicit definition of the two operands. The simple formula ($f_{csg} = \max(f_{sphere}, -f_{knot})$) is evaluated on-the-fly per sample during the ray-casting process.

6.4.3 Applications

This section illustrates several powerful example applications of our rendering pipeline.

Modeling. The high-quality and real-time performance of our rendering pipeline make it ideal for rendering front-end for modeling with implicits, e.g. level-set methods [MBWB02]. As a simple example modeling application, we have implemented CSG operations on distance fields. See Figure 6.17 for an example. For the intersection and subtraction of two objects A and B, the implicit definition is given as $min(f_A - \rho_A, f_B - \rho_B)$ and $min(d_A - \rho_A, \rho_B - f_B)$, respectively. When sampling the volume of one operand, the combined distance field is computed on-the-fly. For each pixel we track which object it belongs to, according to the comparison of the two distance values. The generated object map allows the computation of accurate surface properties and sharp edges on the object resulting from the CSG operation in real-time.

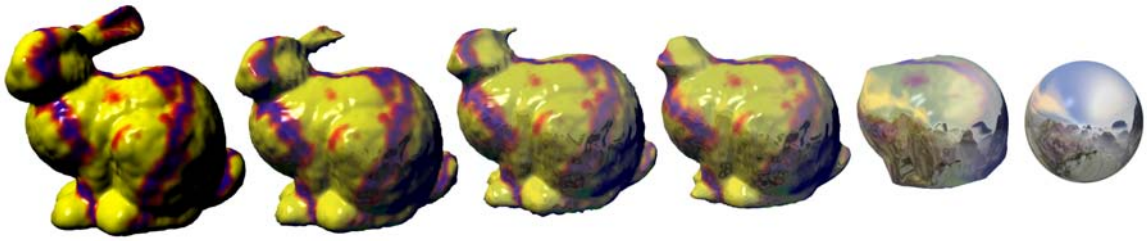


Figure 6.18: Morphing two objects represented as signed distance fields. For this example, simple linear interpolation between the two fields was used. However, all methods that can update a distance field in real-time can be combined with our rendering pipeline.

Morphing. As an example application of morphing one object into another, we simply interpolate linearly between two distance fields and display the result on-the-fly, see Figure 6.18. Higher-quality morphing techniques, e.g. variational implicit surfaces [TO99], can be combined with our technique by establishing a distance field prior to rendering.

Volume rendering. Since the input to our rendering pipeline is an arbitrary scalar field, it is naturally applicable to the rendering of isosurfaces in CT scans. We have integrated our renderer into an existing volume rendering framework as high-quality isosurface rendering front-end. In particular, real-time curvature estimation can be used to guide volume exploration, e.g. visualizing isosurface uncertainty, as has been proposed previously for off-line volume rendering [KWTM03] (see Figure 6.16). Time-dependent volume data can immediately be handled as well.

6.4.4 Performance Evaluation and Discussion

The performance numbers of our rendering pipeline corresponding to the figures shown in this chapter are found in Table 6.1. Except for very small volumes, the overall performance is dominated by the initial volume sampling step that computes approximate intersection positions. Although differential surface properties are expensive to compute in general, the fact that all of these computations have image space complexity combined with fast filtering significantly decrease their impact on the overall frame rate. More importantly, the time spent on these computations is constant with respect to sampling rate and volume resolution. The same is true for intersection optimization via bisection. Table 6.2 illustrates the performance impact of different sampling rates. With respect to adaptive sampling, we compare constant sampling rates with the same rates for the maximum sampling rate r_1 that is used close to the isosurface (Section 6.3.3). For global ray segments, the overhead of the texture indirection introduced by bricking is significant. The indirection is not required for the blocked rendering approach, because the texture offset can be calculated per ray segment. The overhead introduced by the ray segment computation is mitigated by fully effective empty space skipping.

data set	grid size	figure	fps
asian dragon	512x256x256	6.2	20.3
asian dragon	512x256x256	6.11	24.0
david head	512x512x512	6.2	15.3
david head	512x512x512	6.12	14.9
david	576x352x1536	6.3	10.3
cube	64x64x64	6.10	29.6
dragon	512x512x256	6.9	11.7

Table 6.1: Performance of the renderings shown in the figures. Frame rates are given in frames per second for a 512x512 viewport. Four bisection steps have always been used, since they do not influence overall performance significantly.

A problem that can be seen in Figure 6.16, is that even when cubic filters are used, the curvature computed on actual scanned data contains visible noise. However, the quality of cubic filters is almost indistinguishable from filters up to order seven [KWTM03]. In any case, it is important to use full 32-bit floating point precision for all GPU computations.

A limitation of our bisection approach for intersection is that in comparison to an analytic root search [DPH+03] or isolation of exactly one intersection [MKW+04], our discrete sampling with fixed step size does not guarantee correct detection of segments with multiple intersections. Furthermore, our bisection search might not find the intersection closest to the camera in such configurations.

adaptive sampling	brick size	sampling rate (adaptive: r_1) 0.25	0.5	1	2	4	8
no	**none**	33.2	29.0	22.7	16.9	12.4	
no	**32**	23.8	19.5	16.1	11.7	7.2	
$r_1 = 8r_0$	**none**			34.6	27.4	20.3	15.2
$r_1 = 8r_0$	**32**			19.2	13.8	10.2	6.9

Table 6.2: Rendering performance in frames per second corresponding to different sampling rates for the asian dragon rendering found in Figure 6.2. Brick caching introduces an additional texture indirection per sample (Section 6.2.2). Adaptive sampling (Section 6.3.3; $n = 8$) with bricking reduces this overhead compared to constant sampling.

Another consideration is whether to use an interpolating filter, such as trilinear interpolation or Catmull-Rom cubic splines, or a smoothing filter such as the cubic B-spline for reconstruction purposes. Possibly, a very good combination would be to use an interpolating filter for value reconstruction, and a smoothing filter for reconstructing derivatives.

Chapter 7

Quadric Rendering

The geometric complexity of implicit surfaces represented by a grid of function values is only limited by the sampling resolution. On the other hand, quadratic surfaces are a simple basic building block to construct more complicated objects. Due to their compact and simple definition, they are widely used in many applications, for example in CAD systems or constructive solid geometry, and are frequently employed for scientific visualization of tensor fields, particle systems and molecular structures.

While high visual quality can be achieved using sophisticated ray tracing techniques, interactive applications typically use either coarsely tessellated polygonal approximations or pre-rendered depth sprites, thereby trading off visual quality and perspective correctness for higher rendering performance. However, the implicit definition of quadratic surfaces allows efficient GPU-accelerated splatting. A tight bounding box of the quadric is computed in the vertex program. This bounding box is rasterized and for each fragment, the ray-isosurface intersection is computed in the fragment program. Both bounding box and ray hit position can be stated as the root of a bilinear form, corresponding to the implicit surface definition in screen space. Using homogeneous coordinates, the rendering approach also supports perspective projections.

We developed a small library to support hardware accelerated quadratic surfaces within OpenGL. Spheres, ellipsoids, cylinders and cones are available as an additional primitive, which can be rendered using just one vertex call. To demonstrate the usefulness of such a library, we implemented a molecule renderer for the common balls-and-sticks and space-filling representations. To prove the simple incorporation into existing designs, shadow maps and post processing filters to enhance silhouettes have been used. These enhanced rendering effects can greatly improve the spatial perception of the molecule.

7.1 Overview

Quadrics are very well-suited for ray-casting because intersections between a line and a quadric are evaluated efficiently by solving a quadratic equation [TL04]. Although quadric surfaces are featured by many graphics APIs, such as OpenGL utility library [WDS99], they are tessellated for rendering because current graphics hardware is optimized solely for triangle-based rasterization. For high-quality surface rendering, a fine tessellation is required to reduce the error of the piecewise linear approximation. However, small triangles produce disproportionate workload at the vertex shader and triangle setup stage.

Employing the programmability of current graphics hardware, it is possible to implement a ray-casting algorithm for quadrics [TL04], thereby enabling hardware acceleration for this basic primitive type. In the transformation stage of the pipeline, the implicit definition is projected into screen space. This allows the computation of a tight bounding box for the rasterization process. The implicit definition in screen space is also used to efficiently evaluate the quadratic equation per pixel in the fragment shader to determine the ray-surface intersection. Once the ray hit position is known, a standard phong shading model [Bli77] can be evaluated to achieve smooth high-quality shading.

By using our simple interface, quadric rendering primitives can be mixed seamlessly with standard primitives, such as triangles. To demonstrate one possible application of this method, the approach has been applied to illustrative molecule rendering to prove its simple integration into a more complex rendering pipeline as well as its superior quality and performance. Several standard atomic models are used for the study and dissemination of molecular structure and function. Space-filling representations and ball-and-stick models are among the most common ones. The research community uses a number of free and commercial programs to render these models, each having a specific trade-off between quality and rendering speed. Real-time rendering approaches that can cope with large models typically use hardware assisted triangle rasterization [HDS96], whereas high-quality images are produced with ray-tracing [DeL02]. Hand drawn illustrations are used to emphasize certain regions of the model [Goo05], but are very time consuming to produce and require artistic talent.

7.2 Splatting of Quadratic Surfaces

This section will explain all steps necessary to render exact per-pixel shaded quadric surfaces under perspective projections. We will first look at one specific implicit definition of the quadric surface, which can be transformed to screen space even under perspective projections. Namely, we can write the quadratic equation as a bilinear form in homogeneous coordinates. The screen space projection of the bilinear form in homogeneous coordinates is equivalent to a basis

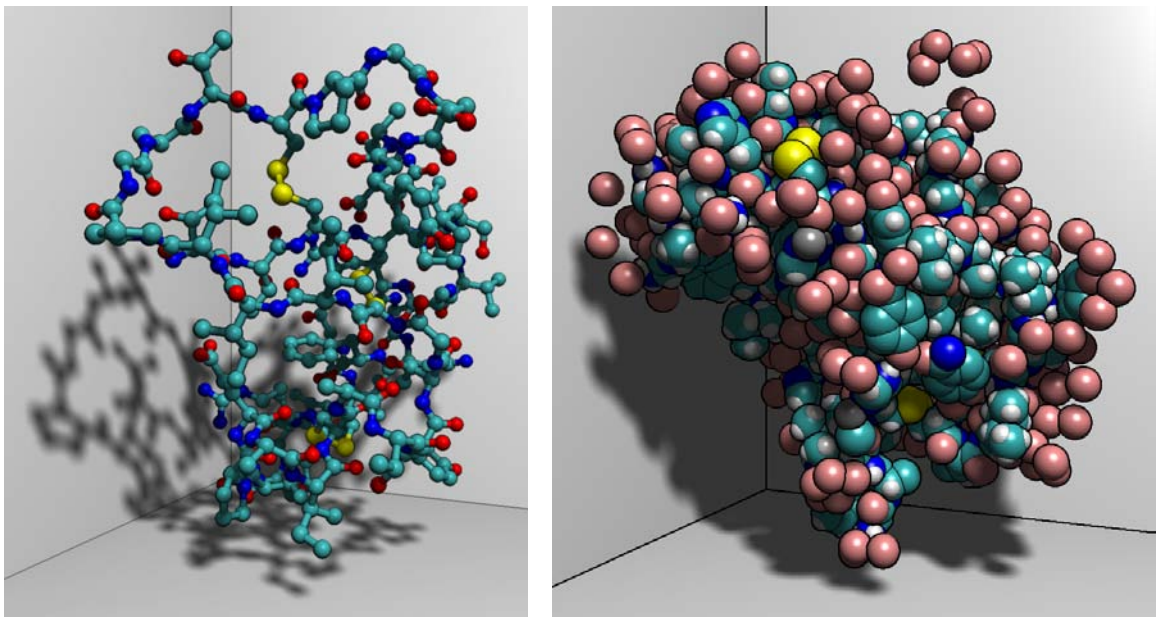


Figure 7.1: Molecular representation of a plant seed protein and human insulin rendered with hardware accelerated sphere and cylinder primitives. Our rendering approach is fast enough to employ silhouette outlining and soft shadows for improved spatial perception at interactive rates.

transformation. This insight will allow the derivation of formulas to compute screen-space bounding boxes, solve ray intersections and evaluate surface normals.

The goal to exert the computation power of GPUs is the main reason to support quadric rendering primitives on this type of hardware. Additionally, it also avoids the tradeoff between rendering quality and excessive triangulation of quadrics in mixed scenes. However, a rendering approach needs to comply with the pipeline of current rasterization APIs [WDS99]. Rendering a triangle with these APIs, such as OpenGL or DirectX, is divided into the following stages: In the vertex shader, each vertex of a triangle is transformed to screen space independently. The rasterization process determines which pixels are covered by the triangle. The color of each of these pixels is then computed in the fragment shader, and written to the frame buffer using the blending stage.

While vertex shader and fragment shaders are fully programmable on current generation graphics cards, all other stages carry out fixed operations. The rasterization process exhibits the biggest limitations for quadric rendering: for triangles, both coverage test and interpolation of vertex attributes are computed using linear functions. For quadratic surfaces however, the coverage test would require a quadratic edge function to be evaluated. The fragment depth is the solution of a quadratic equation. Rasterization hardware is highly optimized for triangles to deal with the impressive throughput of the fragment shaders. Therefore, we can-

not expect to see programmable rasterization supporting quadratic functions in the near future. Instead, we rasterize a bounding polygon of the quadric and execute the exact coverage test in the pixel shader [BK03].

As previously mentioned, vertices are transformed to screen space independently. Therefore, it is not possible to share the transformation of the quadric surface across multiple vertices of the bounding polygon. To avoid recalculation, we would like to use as little vertices as possible for a tight bounding polygon of the quadratic surface. The point sprite, which is a square in screen space defined by a single vertex, provides the best tradeoff of all standard rendering primitives [BK03]. The position and size of the point sprite can be computed in the vertex program to tightly fit the projected quadric.

7.2.1 Implicit Definition in Homogeneous Coordinates

In general, quadratic surfaces are defined as the set of roots of a polynomial of degree two:

$$\begin{aligned} f(x,y,z) = Ax^2 + 2Bxy + 2Cxz + 2Dx + Ey^2 \\ + 2Fyz + Gy + Hz^2 + 2Iz + J = 0 \end{aligned} \tag{7.1}$$

The shape of the quadric is solely determined by the coefficients A through J. The first step is to simplify this longish equation to a compact form. The quadratic equation can be written as a bilinear form in homogeneous coordinates with the *conic matrix* $\mathbf{Q}$:

$$\mathbf{x}^T\mathbf{Q}\mathbf{x} = 0 \qquad \mathbf{Q} = \begin{bmatrix} A & B & C & D \\ B & E & F & G \\ C & F & H & I \\ D & G & I & J \end{bmatrix} \qquad \mathbf{x} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{7.2}$$

This form is not only much shorter, it is also invariant under perspective projections. The key observation is that perspective projections become a linear transformation in homogeneous coordinates. Therefore, the quadric can still be defined in the same form when it is projected to screen space. In general, the implicit definition can be transformed to other coordinate systems by a basis transformation of the bilinear form.

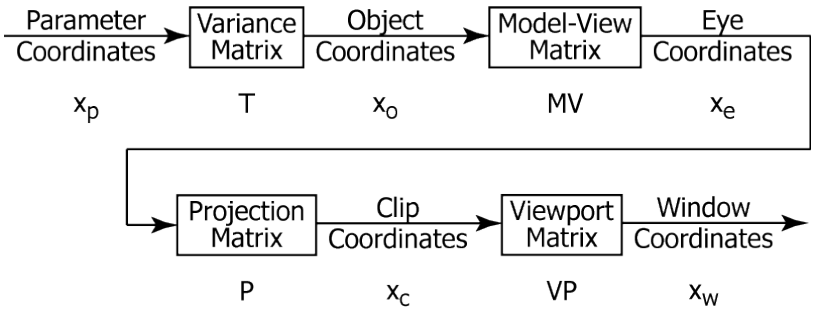


Figure 7.2: The OpenGL vertex transformation sequence is preceded by an additional transformation from parameter coordinates to object coordinates. In parameter coordinates, the conic matrix defining the quadric surface is a diagonal matrix.

7.2.2 Perspective Projection

In OpenGL terminology, the quadric is defined in object space and then subsequently transformed to window coordinates by the transformation sequence denoted in Figure 7.2. Each linear transformation in the sequence is determined by a matrix $\mathbf{M}$, expressing the basis of the previous coordinates in the new coordinate system. In order to transform the bilinear form to the new basis, its corresponding conic matrix $\mathbf{Q}$ needs to be multiplied by the inverse transformation matrix from both sides.

$$x'^T\mathbf{Q}'x' = x'^T(\mathbf{M}^{-\mathbf{T}}\mathbf{Q}\mathbf{M}^{-\mathbf{1}})x' = (x'^T\mathbf{M}^{-\mathbf{T}})\mathbf{Q}(\mathbf{M}^{-\mathbf{1}}x') = x^T\mathbf{Q}x \tag{7.3}$$

Indeed, transforming the conic matrix to a new basis is equivalent to transforming its operands back to the old basis. This allows the bilinear form to be expressed in any coordinate system of the transformation sequence.

For each quadric surface, there is one distinct basis which has been used to simplify the formulas for bounding boxes and equations of ray-surface intersection. Due to the fact that the conic matrix $\mathbf{Q}$ is symmetric, it can be put into diagonal form by a basis transformation. Each non-zero matrix element can be normalized by scaling the basis.

$$\mathbf{T}^T\mathbf{Q}\mathbf{T} = \mathbf{D} \qquad \mathbf{D} \text{ diagonal}, d_{ii} \in \{0, \pm 1\} \tag{7.4}$$

The coordinate system where the bilinear form of a quadric has this diagonal normalized form will be denoted *parameter space*. The transformation matrix $\mathbf{T}$, called *variance matrix*, expresses the basis of the parameter space in object coordinates. The columns contain the axis u, v, t and center c of the quadric, according to Table 7.1.

$$\mathbf{T} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{t} & \mathbf{c} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.5}$$

In the transformation sequence of Figure 7.2, the parameter space is placed in front of the object coordinates. Therefore, every quadric can be defined as an affine transformation of one of the basic classes of quadrics in parameter space. The class of the quadratic surface is determined by the normalized diagonal form.

7.2.3 Bounding Box Computation

As explained in Section 7.2, the rendering approach needs to comply with the graphics pipeline implemented on graphics cards. Rasterization hardware only supports primitives with piecewise linear edges. Therefore, a bounding polygon of the quadric in screen space needs to be computed. Pixels which are rasterized but are not covered by the quadric are culled during fragment shading by evaluating the correct quadratic edge function.

Obviously, the smaller the number of pixels culled in the fragment shader the better. On the other hand, complex bounding polygons should be avoided, because computations cannot be shared across the vertices of the polygon. Therefore, the conic matrix needs to be transformed for each vertex, for example. The point sprite primitive provides the best tradeoff between vertex count and pixel overdraw for most quadrics rendered. Point sprites are rendered with one single vertex call, which completely avoids re-computations. The ratio of culled pixels is usually acceptable. Unfortunately, for thin shapes such as long cylinders, point sprites are not quite optimal.

Note that only ellipsoids are naturally bound by their implicit definition. All other quadrics, such as cylinders or cones, are clipped by the unit cube in parameter space. Bounding box computation will be explained for ellipsoids first and then be generalized for other classes of quadrics. We will use the notion from Figure 7.2 to denote the coordinate systems of vectors and the transformation matrices between them.

To define the parameters of the point sprite, the vertex program needs to compute the center position in clip coordinates and the point sprite radius in window coordinates. A tight bounding box of the projected quadric in clip coordinates is computed first. An axis aligned bounding box in clip coordinates is defined by four intersecting half spaces. Each halfspace is defined by an equation of the following form:

$$\mathbf{x}_c^T \mathbf{n}_c \leq 0 \tag{7.6}$$

For example, the halfspace to the left of b_x is given by $x_c \leq b_x$, corresponding to $\mathbf{n}_c = [1\ 0\ 0\ -b_x]^T$. For the bounding box to be tight, the bounding plane of the halfspace needs to touch the quadric. This condition can be enforced in parameter space, where the quadric is defined by the normalized diagonal matrix $\mathbf{D}$. In parameter space, the ellipsoid coincides with the $\mathbf{S}^2$ sphere, and each point on the

sphere is also the normal of a tangent plane. Therefore, the touching condition in parameter space becomes

$$\mathbf{n}_p^T \mathbf{D} \mathbf{n}_p = 0. \tag{7.7}$$

Transforming this condition to clip space is slightly different than transforming the conic equation, because we are dealing with planes instead of points. Plane normals are transformed with the inverse-transposed matrix.

$$\begin{aligned} \mathbf{x}_p^T \mathbf{n}_p &= \mathbf{x}_p^T (\mathbf{P} \cdot \mathbf{MV} \cdot \mathbf{T})^T \mathbf{n}_c \leq 0 \\ \Rightarrow \quad \mathbf{n}_p &= \quad (\mathbf{P} \cdot \mathbf{MV} \cdot \mathbf{T})^T \mathbf{n}_c = \mathbf{r}_1 - b_x \mathbf{r}_4 \end{aligned} \tag{7.8}$$

with $\mathbf{r}_i$ being the i-th row of the compound transformation matrix $\mathbf{P} \cdot \mathbf{MV} \cdot \mathbf{T}$. Substitution into the constraint 7.7 yields a quadratic equation for the horizontal bounding box coordinate b_x.

$$(\mathbf{r}_4^T \mathbf{D} \mathbf{r}_4) b_x^2 - 2(\mathbf{r}_1^T \mathbf{D} \mathbf{r}_4) b_x + (\mathbf{r}_1^T \mathbf{D} \mathbf{r}_1) = 0. \tag{7.9}$$

The two solutions of the quadratic equation correspond to the position of the left and right border of the bounding rectangle. For the vertical borders, we have to replace $\mathbf{r}_1$ with $\mathbf{r}_2$ in Equation 7.9.

The vertex position of the point sprite in clip coordinates coincides with the center of the bounding box. In homogeneous coordinates, the center position can be stated without division as

$$\mathbf{v}_c = [\mathbf{r}_1^T \mathbf{D} \mathbf{r}_4, \mathbf{r}_2^T \mathbf{D} \mathbf{r}_4, 0, \mathbf{r}_4^T \mathbf{D} \mathbf{r}_4]^T \tag{7.10}$$

The z-coordinate can be set arbitrarily because the depth value is overwritten in the fragment program anyway. The bounding box size has to be defined in terms of the point sprite radius. This term is misleading, because the point sprite is a square, and the radius corresponds to half the edge length in pixel units. Therefore, we apply the viewport transformation to the bounding box size and set half of the larger value to the point size radius.

Point sprite parameters for cylinders and cones can be computed with a similar approach. Those quadrics are culled at the unit cube in parameter space and thus, they are cut off by ellipsoidal caps. One bounding box is computed per elliptic cap and the point sprite is constructed to cover both bounding boxes. Overall, the vertex program for cylinders and cones is slightly larger.

7.2.4 Ray-Quadric Intersection

The rasterization process initiates a fragment shader call for each pixel inside the point sprite. The task of the fragment shader is to kill fragments which are not covered by the quadric and evaluate a lighting model for all others. In order to combine quadrics with standard primitives, the depth value of the surface needs to be computed per pixel as well. For the corresponding ray intersection problem, we

need to find the root of a quadratic equation. If the equation has no real solution, the ray is not intersecting the quadric and the fragment can be killed.

We use the unknown depth value z_w of the intersection to parametrize the viewing ray corresponding to the pixel $[x_w, y_w]$ in window coordinates:

$$\mathbf{x}_w = \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \mathbf{x}'_w + z_w \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \tag{7.11}$$

where x'_w denotes the front plane position $[x_w, y_w, 0, 1]^T$. We take a similar approach as in the last section by transforming the ray equation to parameter space.

$$\mathbf{x}_p = (\mathbf{VP} \cdot \mathbf{P} \cdot \mathbf{MV} \cdot \mathbf{T})^{-1} \mathbf{x}_w = \mathbf{x}'_p + z_w \mathbf{c}_3 \tag{7.12}$$

where $\mathbf{x}'_p$ is the front plane position in parameter coordinates and c_3 is the third column of the inverse transformation matrix. Inserting the ray equation into the quadric definition reveals the formula for the intersection depth in window coordinates:

$$\begin{aligned} 0 &= (\mathbf{x}'_p + z_w \mathbf{c}_3)^T \mathbf{D} (\mathbf{x}'_p + z_w \mathbf{c}_3) \\ &= (\mathbf{c}_3^T \mathbf{D} \mathbf{c}_3) z_w^2 + 2(\mathbf{x}_p'^T \mathbf{D} \mathbf{c}_3) z_w + \mathbf{x}_p'^T \mathbf{D} \mathbf{x}'_p \end{aligned} \tag{7.13}$$

If the discriminant of the quadratic equation is negative, there is no intersection and the fragment can be killed. If the polynomial has two real roots, the smaller one corresponds to the closer intersection and is written to the depth buffer.

Note that the matrix in Equation 7.12 as well as the last summand in Equation 7.13 can be computed per quadric in the vertex program. Several other optimizations are possible to reduce the fragment program length to a small number of operations. The vertex and fragment programs for rendering spheres are listed in Appendix B.

The evaluation of the lighting model requires the position and normal of the surface in eye space. While the position is computed by transformation from window coordinates, the normal is transformed from parameter space:

$$\mathbf{p}_e = (\mathbf{VP} \cdot \mathbf{P})^{-1} \mathbf{x}_w \ , \tag{7.14}$$

$$\mathbf{n}_e = (\mathbf{MV} \cdot \mathbf{T})^{-T} \mathbf{n}_p \ . \tag{7.15}$$

Notice that in order to compute Equations 7.12, 7.14, 7.15, only the quadric-dependent matrix $(\mathbf{MV} \cdot \mathbf{T})^{-1}$ needs to passed from the vertex shader to the fragment shader, since the matrix $\mathbf{VP} \cdot \mathbf{P}$ is constant. Passing too many parameters to the fragment shader reduces performance of the rasterization process. Based on $\mathbf{p}_e$ and $\mathbf{n}_e$, any lighting model can be evaluated on a per-pixel basis. We implemented standard phong shading to prove the rendering quality of our method. Figure 7.3 shows a screen shot of three simple quadratic surfaces.

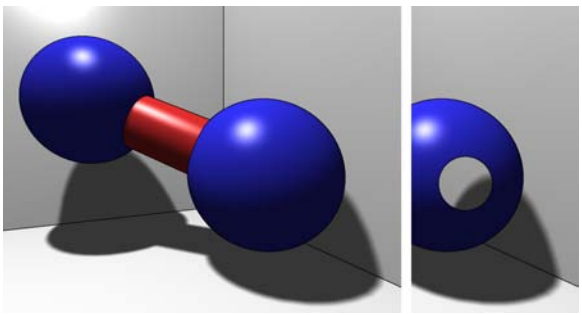


Figure 7.3: A handle consisting of two spheres and a cylinder. Note the smooth highlights as a result of per pixel lighting, as well as the sharp intersection curve due to per-pixel depth correction. In the right image, the near plane has been adjusted to cut off the tip of the right sphere.

7.3 Molecule Rendering

The previous section explained the approach of hardware accelerated quadratic surface rendering. In this section, we are going to present molecule rendering as one application where this technique proves to be advantageous. Illustrations are widely used for the study of molecular structure and function. In order to create these illustrations, several metaphors have been employed.

Balls-and-sticks models are highly effective for displaying the covalent structure of the molecule. Each atom is represented by a sphere and each pair of bonded atoms is connected with a cylinder. This metaphor is particularly useful for organic compounds, because the natural rules of covalent bonding are represented with consistent bond lengths, angles, and geometries. On the other hand, the properties of the electrons are best captured with space-filling representation. A sphere is placed at each atom center with a radius corresponding to the contact distance between atoms. Additional information may be layered onto these representations by coloring the bonds, or by varying the size or texture of cylinders and spheres. Higher level metaphors to capture the topology of biomolecules use ribbons and tubes to represent protein chains and nucleic acids.

Several effective tools have been developed to render molecules in real time using these metaphors. For large structures consisting of several thousands of atoms, real-time visualization of all atoms and covalent bonds poses a challenge because the models need to be tessellated for rasterization. Usually, visual quality is sacrificed for mesh sizes which allow real-time rendering. Using our hardware accelerated algorithm for quadric primitives, tessellation of spheres and cylinders is no longer necessary. Therefore, we can achieve interactive frame-rates for large models consisting of hundreds of thousands of atoms.

Perception of form and shape are crucial for gaining insight into the structure and function of molecules. Several techniques are employed to grasp the three-dimensional shape of a molecule from a flat image. Real-time rendering ap-

proaches exploit the ability of the human mind to decode temporal coherence of rotating objects. Illustrations in printed media use shadows and silhouette edges to improve depth perception. Finally, stereoscopic imaging also creates an artificial impression of three-dimensional vision. Our rendering approach is able to combine all three of these techniques in order to provide keen insight into a molecule.

Hardware support for quadric rendering primitives allows large models consisting of thousands of spheres and cylinders to be rendered in real-time. We implemented shadow maps using percentage-closer filtering for smooth shadow edges. The complexity of expensive per-pixel shading is kept linear with the output resolution using a deferred shading approach. This also allows us to apply post processing filters for silhouette outlining. The improvement in spatial perception when using these effects is depicted in Figure 7.4 and 7.7. Finally, support for stereoscopic rendering is available for most OpenGL drivers.

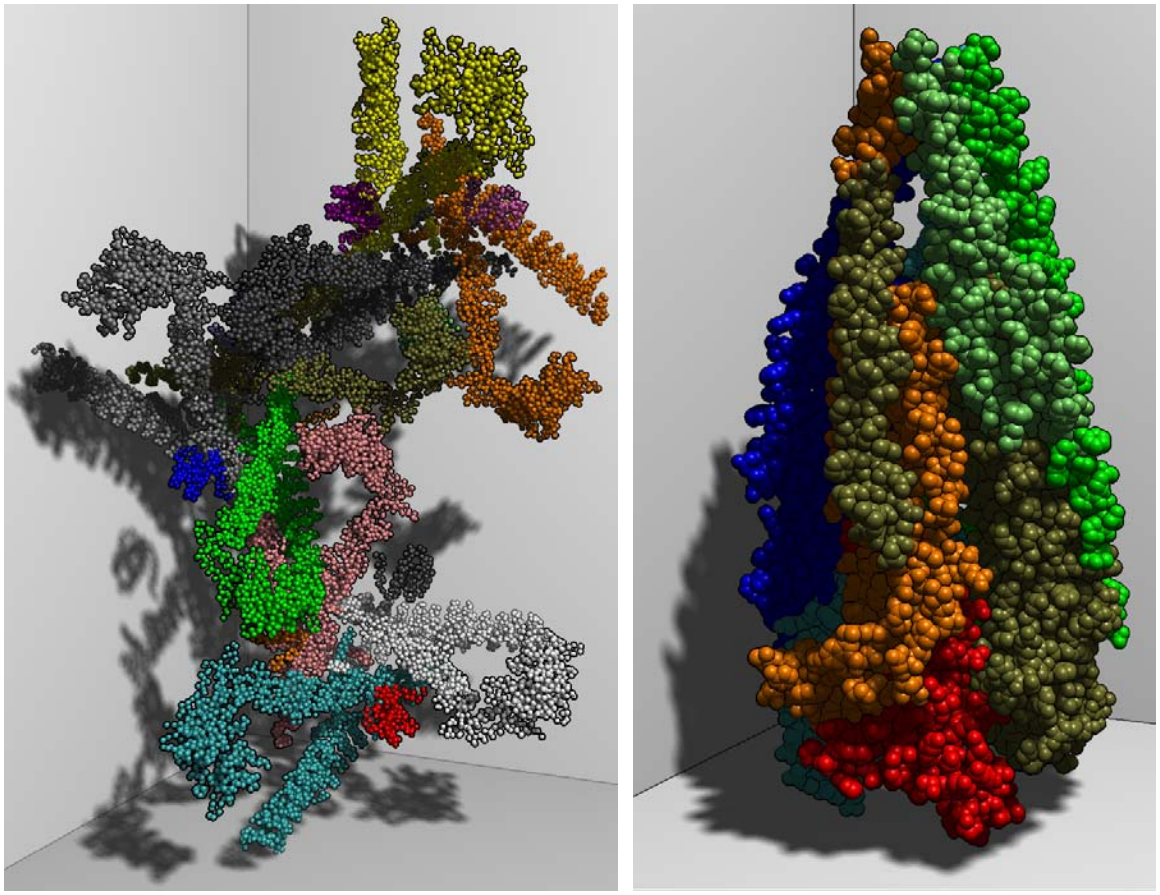


Figure 7.4: Improved spatial perception with visual effects. In comparison to pure per-pixel phong shading, silhouette and crease outlining and soft shadows make it much easier to conceive the structure of the molecule.

7.3.1 Deferred Shading

Similar to the rendering approach presented in Chapter 6, a range of effects are used to compute the color of a quadric surface fragment, including soft shadows and silhouette edge detection. To avoid expensive evaluation of the shading equation for fragments which are subsequently overdrawn by another fragment closer to the camera, deferred shading [DWS+88] is employed again (see Section 6.4). The key idea is to write all parameters required to evaluate the shading equation for a fragment to a screen-sized *geometry buffer*. When the entire scene for a frame is drawn, the shading equation is evaluated only once for each pixel using the parameters of the buffer.

As a result of employing shadow mapping, our pipeline consists of three separate rendering passes. In the first pass, the scene is rendered from light view to a depth map, called *shadow map*. The light view is defined by the bounding frustum of the spot light illuminating the scene.

In the second pass, the scene is rendered from the camera view. Instead of evaluating the shading equation for each fragment, we write the following parameters to the geometry buffer: fragment depth in window coordinates, surface diffuse color and surface normal in eye coordinates.

In the final pass of the pipeline, the color for each pixel is determined using the parameters of the geometry buffer. The fragment depth read from the buffer and the implicit pixel position reveal the window coordinates of the fragment. This position can be transformed to eye space to evaluate the local phong lighting equation, as well as to light space to perform shadow map lookups. Finally, an edge detection filter is applied to normal map and depth map to detect silhouette edges. The outlines are then composited with the shaded scene.

The complete pipeline as well as the buffers of a sample scene are depicted in Figure 7.5.

7.3.2 Soft Shadow Maps

Shadows provide valuable information about the spatial relationship of groups of atoms that form a molecule. Unfortunately, shadowing imposes a significant workload on a rendering algorithm because the scene needs to be rendered multiple times, once for every light source and once for the final image generation. Therefore, shadow techniques such as shadow volumes or shadow maps are usually too slow for real-time rendering of molecules. Our hardware algorithm for quadric primitives is fast enough to provide room for such visual effects even for large molecular structures. Soft shadow maps using percentage-closer filtering [RSC87] have been implemented.

Standard shadow map approaches suffer from unnatural hard or aliased shadow boundaries. Thus, they often overstrain the human eye instead of providing helpful information about the spatial relationship. Shadows from area lights with soft

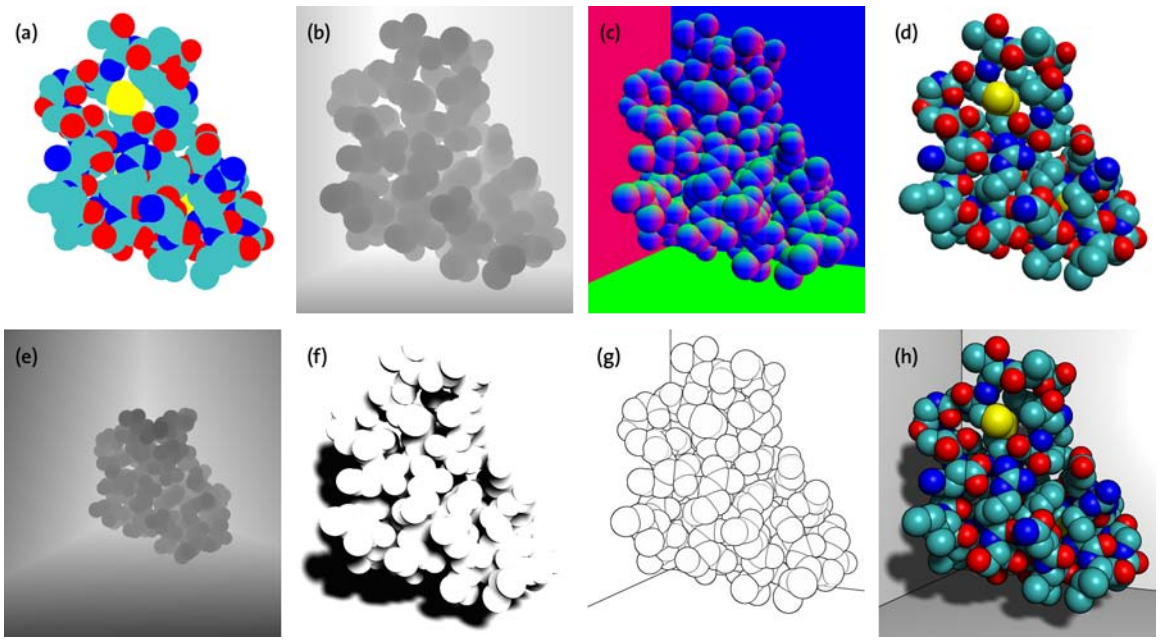


Figure 7.5: Different buffers and effects generated for each frame of the deferred rendering pipeline. Three geometry buffers store the parameters of the shading equation: (a) diffuse color, (b) fragment depth, (c) surface normal. Per-pixel phong lighting (d) is then evaluated for each pixel using the geometry buffers. A shadow map is rendered from light view (e) and filtered to generate smooth shadow edges (f). Silhouette and crease outlines (g) are extracted by an edge detection filter applied to (b) and (c). Shadows and outlines are composited with the phong lighting to generate the final image (h).

edges are much more subtle, but are much harder to compute. Sufficiently large percentage-closer filters [RSC87] approximate penumbras which are formed when part of the light source is visible from a surface element.

The basic idea is to convolve the shadow function with a radial box filter to blur the hard shadows edges [UA04]. Shadow maps cannot be pre-filtered, because filtering has to occur after comparison of fragment depth and shadow map. The convolution is approximated by a finite sum using Monte-Carlo sampling. A regular grid of samples in polar coordinates is jittered using a pseudo-random offset table. Each pixel, modulo a small tile size, uses a different offset table to achieve locally unique and constant jittering.

A moderate number of 64 samples is required to avoid visible noise inside the penumbra region. However, the convolution is useless in areas which are completely in shadow or completely visible to the light source. Therefore, the first 8 samples are tested to determine whether the current fragment is inside a penumbra region. Otherwise the sampling process is stopped. In combination with our deferred shading approach, which evaluates the filtered shadow map only once per pixel, this optimization allows high-quality soft shadows in real-time.

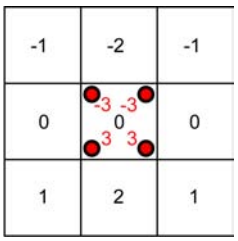


Figure 7.6: Evaluation of Sobel edge detection filter. The stencil of the gradient filter (black) in vertical direction can be evaluated with four fetches (red, 2:1 ratio between texels) using linear texture filtering. A rotated weighting of the same values yields the horizontal derivative.

7.3.3 Post Processing Filters

Object and crease outlines [MBC02] are a non-photorealistic visual cue for distinguishing neighboring objects of similar color. Silhouettes and creases can be detected by employing a post processing filter. The gradient magnitude of the depth map measures the presence of silhouettes. The gradient magnitudes of all normal map components are interpreted as a vector and its length measures the presence of creases. Note that the generation of both normal and depth map is already required for deferred shading.

For each component of the normal map and the depth map, the gradient length is computed using a standard Sobel edge detection filter. For this, two 3x3 convolution masks are applied to each pixel, to evaluate horizontal and vertical derivatives. We use the technique described in Chapter 5 to take advantage of the bi-linear texture filter mode to reduce the amount of texture samples required to evaluate the convolution sum. With intermediate sampling positions shown in Figure 7.6, the gradient length can be evaluated with just four texture samples.

The intensity of silhouettes and creases measured by the edge detection is mapped to a blending weight using clamped linear ramps. To determine the color of the final pixel, the outline color is then blended with the diffuse surface color.

7.4 Results

We implemented a simple extension to the OpenGL API to render quadratic surfaces. Table 7.1 gives an overview of the primitive that we support and the parameters to define their shape. Note that some parameters are redundant with respect to affine transformations of the standard modelview matrix. For example, translating a sphere to a new center or stretching it to an ellipsoid is also possible by corresponding calls to `glTranslate`, `glScale` and `glRotate`. However, changing the transformation matrix has a large driver overhead or even flushes the pipeline. Instead, we use vertex attributes to specify scale, translation and rotation of quadrics, enabling the use of efficient vertex arrays.

The raw vertex and fragment throughput of our implementation is stated in Table 7.2. The vertex throughput determines how many quadric primitives can be rendered per second. Note that the rate depends on the quadric type, because

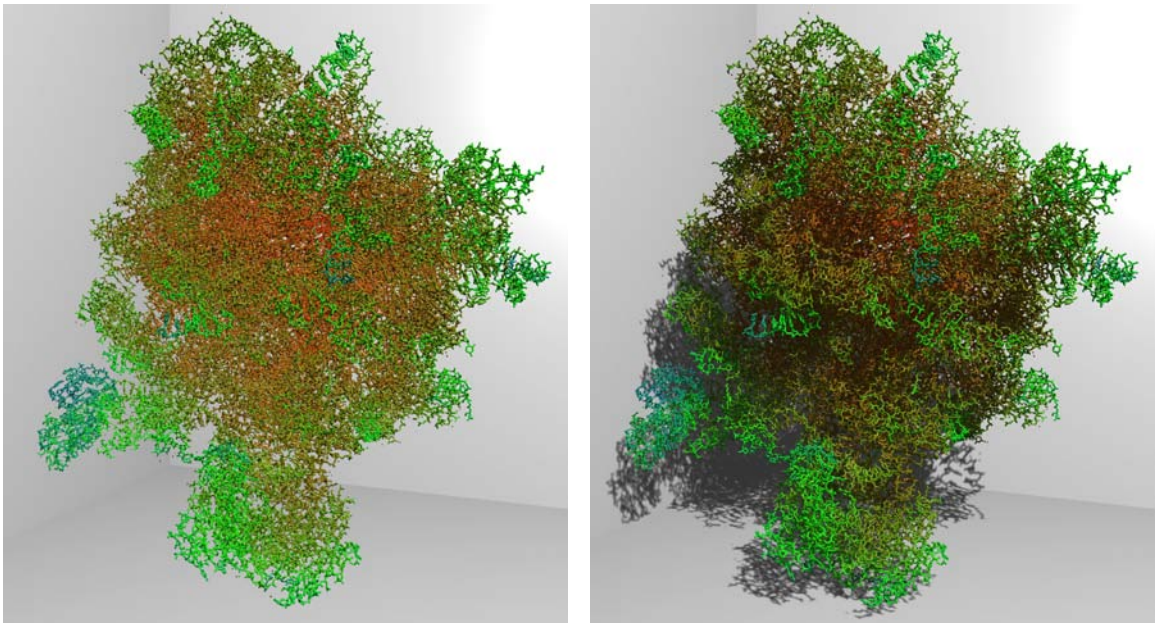


Figure 7.7: A ball-and-stick model consisting of 99k spheres and 198k cylinders. Compared to simple shading (left), our superior quality visualization (right) greatly improves the spatial perception, and can still be rendered at 13 fps (1024×768 res.).

bounding box evaluation is more involved for cylinders and cones. On the other hand, the rate of fragment rasterization measures how many pixels of quadric surface elements can be rendered per second.

Table 7.3 compares the rendering performance of different kinds of shading on several molecules of varying complexities. Using standard *direct shading*, the shading equation is evaluated multiple times for a single pixel when one surface is rendered on top of another one. In contrast, deferred shading evaluates the shading equation in a post-processing step exactly once per pixel at the additional cost of storing the shading parameters in offscreen buffers. While for simple shading models the direct approach is faster, the deferred technique clearly pays off for more complex shading including soft shadows and silhouette lines.

We also examined the relative workload of each individual rendering effect in our deferred shading pipeline: the initial shadow map generation is 12%, the per-pixel Phong shading 35%. Filtering the shadow map with 64 samples for smooth shadow edges is clearly the most time consuming element (45%). It is divided in an 8-sample-test to determine whether pixels lie in the penumbra region (19%) and the full 64-sample filtering for those pixels that do so (26%). Including the silhouette and crease lines then adds the missing 8%. Performing full shadow map filtering only in regions that are tested to lie inside a penumbra region results in an 31% rendering speed-up on average for our sample scenes.

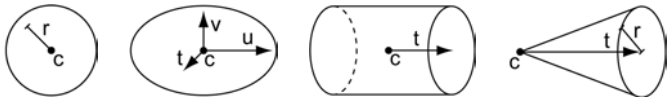
primitive type	sphere	ellipsoid	cylinder	cone
center	**c**	**c**	**c**	**c**
radius	**r**			**r**
axis		**u,v,t**	**t**	**t**
				

Table 7.1: Quadric primitive types. Each primitive can be rendered with one single vertex call. The center is specified as the vertex position and the remaining parameters are stored in the texture coordinates. From these parameters, the vertex program generates a point sprite which bounds the primitive in screen space. The fragment program then computes the exact coverage and performs per-pixel lighting of the quadric.

primitive type	sphere	cylinder	points
primitives (s^{-1})	43.5M	21.0M	59.3M
fragments (s^{-1})	178M	162M	266M

Table 7.2: Performance of our quadric primitive rendering algorithm. Both primitive and fragment rates are measured independently on a NVIDIA GeForce 6800GT. The right column states the values for OpenGL points with per-pixel lighting for comparison.

figure	#spheres	#cylinders	direct shading		deferred shading	
			phong	+SM	phong	+SM+SL
7.8	99k	198k	37 fps	3.9 fps	34 fps	13 fps
7.7	52k	–	31 fps	2.6 fps	26 fps	15 fps
7.4, right	15k	–	79 fps	5.5 fps	51 fps	22 fps
7.1, right	1712	–	94 fps	7.2 fps	67 fps	22 fps

Table 7.3: Performance comparison for standard *direct shading* and *deferred shading* at a viewport resolution of 1024×768 on a P4, 2.4GHz, with GeForce 6800GT. While for simple Phong shading without shadows the direct shading is faster, the deferred shading is clearly superior when adding soft shadows (+*SM*) and silhouette and crease lines (+*SL*).

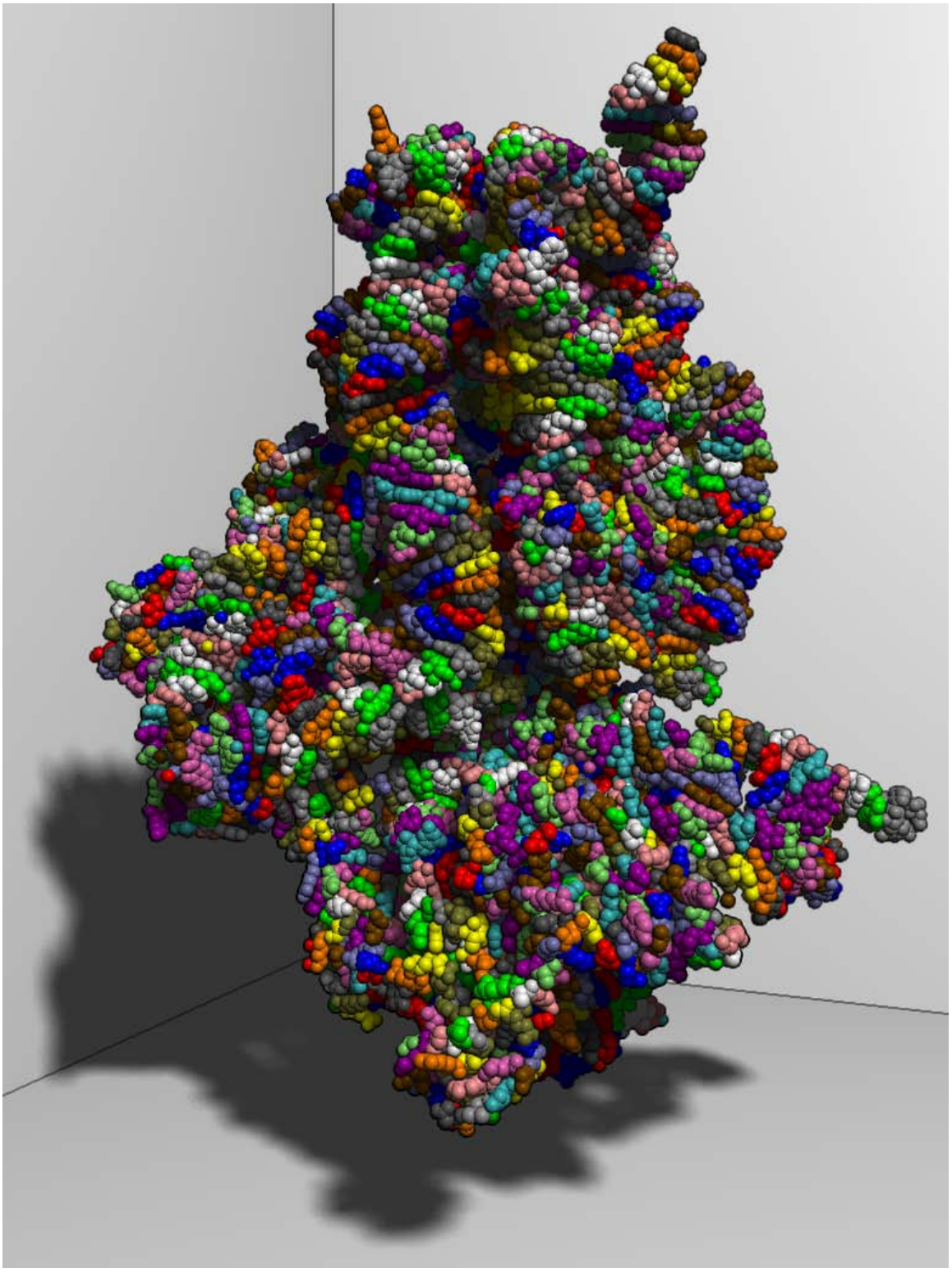


Figure 7.8: This complex molecular structure consists of 52k atoms and can be rendered at 15 fps including soft shadow maps and silhouette contouring (1024×768 resolution).

Chapter 8

Conclusion

This chapter consists of a summary of the methods and results presented in the preceding sections of this thesis. The work is concluded by pointing out possible directions for future work in this field.

8.1 Summary of Contributions

The focus of this thesis is the representation of implicit surfaces and methods for real-time rendering using consumer graphics hardware. In this context, the following contributions have been made:

- A hashed representation of the octree for adaptive sampling has been presented. As a sorted linear traversal of the hierarchical space subdivision is not required for processing implicit surfaces, hashing can provide a more efficient representation in terms of memory consumption and performance.
- The kD-tree used for irregular space subdivision has been extended to provide finer grained classification of its elements. By using small overlaps of sibling nodes, elements that would have been intersected by a single split plane can be assigned to one of the subtrees, thereby allowing tighter bounding volumes. Only minor modifications are required for algorithms employing the kD-tree for range and proximity queries but the number of culled elements can nonetheless be significantly increased.
- An implementation of the signed distance transform employing graphics hardware has been introduced. A signed distance field in the proximity of a triangle mesh is computed by scan converting a set of polyhedra associated with the mesh primitives. The scan conversion is accelerated by the hardware-native polygon rasterization. The transfer bottleneck of geometry data is mitigated by simple polyhedra, which also reduce the average amount of overlap.

- Various methods for reconstructing a continuous representation from a set of surface samples have been analyzed. A combination of these methods has been presented which accurately approximates the distance to a smooth surface interpolating the given point cloud. Smooth blending of local first-order approximations near the surface has been combined with the distance to the closest sample in areas where value extrapolation fails. A scale-invariant formula for the distance to a regular sampling has been used to determine the blending weight between the two approximations.

- An algorithm for texture filtering with a cubic B-Spline kernel and its first and second derivatives on graphics hardware has been presented. In order to reduce the large amount of texture samples required in two and three dimensions, cubic filtering has been built on hardware-native linear texture fetches. Because linear filtering is as fast as nearest neighbor sampling but can compute a convex combination of up to eight texels, high-quality filtering with a kernel of 64 texels is possible with a moderate number of 8 texture fetches. The approach can also be employed for other filter operations in order to reduce the number of texture lookups and value interpolations as long as the neighboring convolution weights have equal signs.

- A pipeline for real-time rendering of implicit surfaces stored as a regular grid has been developed. A hardware accelerated ray-casting approach was used to perform direct rendering on a two-level hierarchical representation which allows to perform empty space skipping and circumvent memory limitations of graphics hardware. Adaptive sampling and iterative refinement lead to high-quality ray-surface intersections. All shading operations were deferred to image space, making their computational effort independent of the size of the input data. Smooth surface normals and extrinsic curvatures were evaluated on-the-fly to perform advanced shading using high-quality lighting and non-photorealistic effects in real-time.

- An algorithm for hardware accelerated rendering of quadratic surfaces such as spheres, ellipsoids, cylinders and cones has been presented. Each primitive is rendered with a single vertex call and rasterized as a point sprite. The implicit definition of the quadric is transformed to screen space to compute a tight bounding box and evaluate ray intersections. A molecule renderer employing soft shadows and silhouette outlining to improve spatial perception was used to demonstrate performance and flexibility of the approach.

We conclude that the computational power of current generation GPUs allow algorithms for representation and rendering of implicit surfaces that can outperform pure software implementations in terms of performance. Although programmability of GPUs has advanced tremendously with every generation, there are still a few features that are missing to efficiently implement fully hierarchical data structures.

8.2 Discussion and Future Work

The intuitive and consistent handling of surfaces with complex topology has proven to be advantageous for modeling and simulation of natural phenomena such as fire, liquid and smoke as well as for a variety of geometric problems like surface reconstruction. However, they are less suited to accurately represent static surfaces for applications in industrial design and manufacturing. Memory constraints and high-performance processing and rendering require a careful design of efficient and compact representations for the scalar function. It has been demonstrated that memory efficient representation and real-time rendering of implicit surfaces can be achieved, especially if the computational power of graphics hardware is employed to process the volumetric data.

Adaptive sampling grids and general hierarchical data structures are the base for a large amount of algorithms in computer graphics with very good performance and memory characteristics, for example ray tracing, adaptive textures and irregular framebuffers, visibility sorting and particle simulation. However, they do not seem to comply very well with the design of current graphics cards, which lack efficiency when employing recurring dependent texture fetches and branching. Several bottlenecks could be responsible, for example the wide pixel pipeline or the way threads are scheduled. The most promising approach would be to investigate the potential of application-specific prefetch and cache policies on new multimedia architectures with software-manageable caches. I expect that many algorithms for hierarchical data structures could benefit from a small common set of strategies to improve reference locality and exploit parallelism. Those strategies would have to be identified and their usefulness would need to be proven for a range of applications. An analysis of the amount of hardware logic required to support these strategies would indicate the feasibility of GPUs to support hierarchical data structures.

Modeling operations for implicit surfaces can be based on Boolean operations or the levelset formulation. However, modeling packages for parametric surfaces include a range of tools to perform large free-form deformations in real-time. Approaches for such modeling operations using implicit surfaces should be investigated since the inconvenient problem of self-intersections could be avoided.

Several methods for processing implicit surfaces employ graphics hardware for better performance. However, the transfer of data from main memory to texture memory is relatively slow. Therefore, it would be beneficial to implement a complete framework for implicit surfaces that performs all computations directly on the graphics card, including surface processing and rendering. Engineering such a complete framework of algorithms presented in this thesis and published by other researchers has not been pursued in this dissertation.

The conversion of triangle meshes to an implicit representation generally relies on the triangle mesh to be a consistent manifold. However, triangle meshes generated by automatic processes often contain small deficiencies such as flipped

or degenerate triangles, which can produce an inconsistent sign in the distance field generated by scan conversion. A mesh validation or even a method to fix the triangulation in such cases should be implemented to avoid erroneous distance fields.

The construction of cubic B-Spline filters based on fast linear filtering could be applied to other filter kernels and applications which require higher-order filters. For example, a filter method for semi-regular grids which avoids discontinuities at hanging nodes would be beneficial for adaptive sampled distance field.

Generalizations of quadratic surfaces can provide an extended range of shapes through a few additional parameters. These superquadrics and supershapes have proven to be a valuable primitive for visualization tasks. Hardware-accelerated, real-time rendering of these primitives could improve the analysis of high-dimensional time-dependent data.

Bibliography

[AS95] David Adalsteinsson and James A. Sethian. A fast level set method for propagating interfaces. *J. Comput. Phys.*, 118(2):269–277, 1995.

[AS99] David Adalsteinsson and James A. Sethian. The fast construction of extension velocities in level set methods. *J. Comput. Phys.*, 148(1):2–22, 1999.

[Bær02] Jakob A. Bærentzen. *Manipulation of volumetric solids with applications to sculpting*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, 2800 Kgs. Lyngby, Denmark, 2002.

[Bar81] Alan H. Barr. Superquadrics and angle-preserving transformations. *IEEE Computer Graphics and Applications*, 1(1):11–23, 1981.

[Bar86] Alan H. Barr. Ray tracing deformed surfaces. In *Proc. of SIGGRAPH '86*, pages 287 – 296, 1986.

[BFA02] Robert Bridson, Ronald Fedkiw, and John Anderson. Robust treatment of collisions, contact and friction for cloth animation. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 594–603, New York, NY, USA, 2002. ACM Press.

[BG97] Rick Beatson and Leslie Greengard. A short course on fast multipole methods. In M. Ainsworth, J. Levesley, W.A. Light, and M. Marletta, editors, *Wavelets, multilevel methods and elliptic PDEs*, pages 1–37. Oxford University Press, 1997.

[BK03] Mario Botsch and Leif P. Kobbelt. High-quality point-based rendering on modern gpus. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, page 335, Washington, DC, USA, 2003. IEEE Computer Society.

[Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198, New York, NY, USA, 1977. ACM Press.

[Bli82] James F. Blinn. A generalization of algebraic surface drawing. *ACM Trans. Graph.*, 1(3):235–256, 1982.

[Blo94] Jules Bloomenthal. An implicit surface polygonizer. In *Graphics gems IV*, pages 324–349. Academic Press Professional, Inc., San Diego, CA, 1994.

[Blu67] Harry Blum. A transformation for extracting new descriptors of shape. In Weiant Wathen-Dunn, editor, *Models for the Perception of Speech and Visual Form*, pages 362–380. MIT Press, Cambridge, MA, USA, 1967.

[BMWM01] David E. Breen, Sean Mauch, Ross T. Whitaker, and Jia Mao. 3D metamorphosis between different types of geometric models. In *Eurographics 2001 Proceedings*, pages 36–48. Blackwell Publishers, September 2001.

[BSK04] Mario Botsch, Michael Spernat, and Leif P. Kobbelt. Phong splatting. In *Proceedings Symposium on Point-Based Graphics*, pages 25–32. Eurographics, 2004.

[Buh03] Martin D. Buhmann. *Radial Basis Functions : Theory and Implementations*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2003. 270 pages.

[BW97] Jules Bloomenthal and Brian Wyvill, editors. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1997.

[Car76] Manfredo P. Do Carmo. *Differential Geometry of Curves and Surfaces*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1976. 503 pages.

[CBC+01] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. Reconstruction and representation of 3D objects with radial basis functions. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 67–76, New York, NY, USA, 2001. ACM Press.

[CMM+97] Paolo Cignoni, Paola Marino, Claudio Montani, Enrico Puppo, and Roberto Scopigno. Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, 1997.

[COSL98] Daniel Cohen-Or, Amira Solomovic, and David Levin. Three-dimensional distance field metamorphosis. *ACM Trans. Graph.*, 17(2):116–141, 1998.

[CSS98] Yi-Jen Chiang, Cláudio T. Silva, and William J. Schroeder. Interactive out-of-core isosurface extraction. In *Proc. of IEEE Visualization '98*, pages 167–174, 1998.

[DCH88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 65–74, New York, NY, USA, 1988. ACM Press.

[DeL02] Warren L. DeLano, 2002. The PyMOL Molecular Graphics System: `http://www.pymol.org`.

[DPH+03] D. DeMarle, S. Parker, M. Hartner, Christiaan Gribble, and Charles D. Hansen. Distributed interactive ray tracing for large volume visualization. In *Proc. of IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 87–94, 2003.

[DvOG04] Daniel Dekkers, Kees van Overveld, and Rob Golsteijn. Combining csg modeling with soft blending using lipschitz-based implicit surfaces. *Vis. Comput.*, 20(6):380–391, 2004.

[DWS+88] M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt. The triangle processor and normal vector shader: a vlsi system for high performance graphics. In *Proc. of ACM SIGGRAPH 88*, pages 21–30, 1988.

[Egg98] Hinnik Eggers. Two fast euclidean distance transformations in z2 based on sufficient propagation. *Comput. Vis. Image Underst.*, 69(1):106–116, 1998.

[EHK+06] Klaus Engel, Markus Hadwiger, Joe M. Kniss, Christof Rezk-Salama, and Daniel Weiskopf. *Real-Time Volume Graphics*. A K Peters, Ltd, 2006.

[EKE01] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proc. of Graphics Hardware 2001*, pages 9–16, 2001.

[Fer05] Randima Fernando. Percentage-closer soft shadows. In *SIGGRAPH '05 Proceedings (sketch)*, August 2005.

[FN80] Richard Franke and Gregory M. Nielson. Smooth interpolation of large sets of scattered data. *International Journal for Numerical Methods in Engineering*, 15(11):1691–1704, 1980.

[FP02] Sarah F. Frisken and Ronald N. Perry. Simple and efficient traversal methods for quadtrees and octrees. *Journal of Graphics Tools*, 7(3):1–11, 2002.

[FPRJ00] Sarah F. Frisken, Ronald N. Perry, Alyn P. Rockwood, and Thouis R. Jones. Adaptively sampled distance fields: a general representation of shape for computer graphics. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 249–254, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[FT01] William H. Ford and William R. Topp. *Data Structures with C++ Using STL*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[GBB03] Johan Gielis, Bert Beirinckx, and Edwin Bastiaens. Superquadrics with rational and irrational symmetry. In *SM '03: Proceedings of the eighth ACM symposium on solid modeling and applications*, pages 262–265, New York, NY, USA, 2003. ACM Press.

[GGSC98] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 447–452, New York, NY, USA, 1998. ACM Press.

[GH91] Tinsley A. Galyean and John F. Hughes. Sculpting: an interactive volumetric modeling technique. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 267–274, New York, NY, USA, 1991. ACM Press.

[Gib98] Sarah F. F. Gibson. Using distance maps for accurate surface representation in sampled volumes. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 23–30, New York, NY, USA, 1998. ACM Press.

[GLDK95] Markus Gross, Lars Lippert, Andreas Dreger, and Rolf M. Koch. A new method to approximate the volume rendering equation using wavelets and piecewise polynomials. *Computers and Graphics*, 19(1):47–62, 1995.

[Goo05] David S. Goodsell. Visual methods from atoms to cells. *Structure*, 13:347–354, 2005.

[Gra93] Gaye Graves. The magic of metaballs. *Computer Graphics World*, 16(5):27–32, 1993.

[Gre04] Simon Green. Procedural volumetric fireball effect. In *NVSDK samples*. NVIDIA Corp., 2004.

[GSLF05] Eran Guendelman, Andrew Selle, Frank Losasso, and Ronald Fedkiw. Coupling water and smoke to thin deformable and rigid shells. *ACM Trans. Graph.*, 24(3):973–981, 2005.

[Gum03] Stefan Gumhold. Splatting illuminated ellipsoids with depth correction. In Thomas Ertl, editor, *Proceedings of VMV 2003*, pages 245–252. Aka GmbH, November 2003.

[GWGS02] Stefan Guthe, Michael Wand, Julius Gonser, and Wolfgang Straßer. Interactive rendering of large volume data sets. In *Proc. of IEEE Visualization 2002*, pages 53–60, 2002.

[HBH03] Markus Hadwiger, Christoph Berger, and Helwig Hauser. High-quality two-level volume rendering of segmented data sets on consumer graphics hardware. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 40, Washington, DC, USA, 2003. IEEE Computer Society.

[HCLL01] Jian Huang, Roger Crawfis, Shao-Chiung Lu, and Shuh-Yuan Liou. A complete distance field representation. In Thomas Ertl, Kenneth I. Joy, and

Amitabh Varshney, editors, *IEEE Visualization*. IEEE Computer Society, 2001.

[HDS96] William Humphrey, Andrew Dalke, and Klaus Schulten. VMD – Visual Molecular Dynamics. *Journal of Molecular Graphics*, 14:33–38, 1996.

[Hec89] Paul S. Heckbert. Fundamentals of texture mapping and image warping. Technical report, University of California at Berkeley, Berkeley, CA, 1989.

[HKG00] Jiří Hladuvka, Andreas König, and Eduard Gröller. Curvature-based transfer functions for direct volume rendering. In *Proc. of SCCG 2000*, pages 58–65, 2000.

[HKL+99] Kenneth E. Hoff III, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. *Computer Graphics*, 33(Annual Conference Series):277–286, 1999.

[HLHS03] Jean-Marc Hasenfratz, Marc Lapierre, Nicolas Holzschuch, and François Sillion. A survey of real-time soft shadows algorithms. *Computer Graphics Forum*, 22(4):753–774, Dec. 2003. State-of-t.

[Hof89] Christoph M. Hoffmann. *Geometric and solid modeling: an introduction*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1989.

[HSS+05] Markus Hadwiger, Christian Sigg, Henning Scharsach, Katja Bühler, and Markus Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. In Marc Alexa and Joe Marks, editors, *Proceedings of Eurographics '05*, volume 24, pages pp303–312, Dublin, Ireland, September 2005. Blackwell Publishing.

[HTHG01] Markus Hadwiger, Thomas Theußl, Helwig Hauser, and Eduard Gröller. Hardware-accelerated high-quality filtering on pc hardware. In *VMV '01: Proceedings of the Vision Modeling and Visualization Conference 2001*, pages 105–112. Aka GmbH, 2001.

[HXP03] Xiao Han, Chenyang Xu, and Jerry L. Prince. A topology preserving level set method for geometric deformable models. *IEEE Trans. Pattern Anal. Mach. Intell.*, 25(6):755–768, 2003.

[IFP95] Victoria Interrante, Henry Fuchs, and Stephen Pizer. Enhancing transparent skin surfaces with ridge and valley lines. In *Proc. of IEEE Visualization '95*, pages 52–59, 1995.

[JBS06] Mark W. Jones, Jakob A. Bærentzen, and Milos Sramek. 3D distance fields: A survey of techniques and applications. *Transactions on Visualization and Computer Graphics*, 2006. (accepted for publication).

[Jen96] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30, London, UK, 1996. Springer-Verlag.

[JLSW02] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of hermite data. In *Siggraph 2002, Computer Graphics Proceedings*, pages 339–346. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2002.

[JQR03] Calvin R. Maurer Jr., Rensheng Qi, and Vijay V. Raghavan. A linear time algorithm for computing exact euclidean distance transforms of binary images in arbitrary dimensions. *IEEE Trans. Pattern Anal. Mach. Intell.*, 25(2):265–270, 2003.

[KB89] Devendra Kalra and Alan H. Barr. Guaranteed ray intersections with implicit surfaces. In *Proc. of SIGGRAPH '89*, pages 297 – 306, 1989.

[KBSS01] Leif P. Kobbelt, Mario Botsch, Ulrich Schwanecke, and Hans-Peter Seidel. Feature-sensitive surface extraction from volume data. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 57–66. ACM Press / ACM SIGGRAPH, 2001.

[KDR+02] Ujval J. Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany. The imagine stream processor. In *Proceedings of the IEEE International Conference on Computer Design*, pages 282–288, September 2002.

[KE02] Martin Kraus and Thomas Ertl. Adaptive texture maps. In *Proc. of Graphics Hardware 2002*, pages 7–15, 2002.

[Kin04] Gordon L. Kindlmann. Superquadric tensor glyphs. In Oliver Deussen, Charles D. Hansen, Daniel A. Keim, and Dietmar Saupe, editors, *VisSym 2004, Symposium on Visualization*, pages 147–154, Konstanz, Germany, May 2004. Eurographics Association.

[KS98] Ron Kimmel and James A. Sethian. Computing geodesic paths on manifolds. *Proceedings of National Academy of Sciences*, 95(15):8431–8435, 1998.

[KW03a] Jens Krüger and Rüdiger Westermann. Acceleration techniques for GPU-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, pages 287–292, Washington, DC, USA, 2003. IEEE Computer Society.

[KW03b] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, 2003.

[KWTM03] Gordon Kindlmann, Ross T. Whitaker, Tolga Tasdizen, and Torsten Moller. Curvature-based transfer functions for direct volume rendering: Methods and

applications. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, pages 513–520, Washington, DC, USA, 2003. IEEE Computer Society.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Proc. of SIGGRAPH '87*, pages 163–169, 1987.

[LCN98] Barthold Lichtenbelt, Randy Crane, and Shaz Naqvi. *Introduction to volume rendering*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

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

[Lev98] David Levin. The approximation power of moving least-squares. *Math. Comput.*, 67(224):1517–1531, 1998.

[Lev03] David Levin. Mesh-independent surface interpolation. In Guido Brunnett, Bernd Hamann, and Heinrich Müller, editors, *Geometric Modeling for Scientific Visualization*, pages 37–49. Springer Verlag, Heidelberg, Germany, 2003.

[LGF04] Frank Losasso, Frédéric Gibou, and Ron Fedkiw. Simulating water and smoke with an octree data structure. *ACM Trans. Graph.*, 23(3):457–462, 2004.

[LHN05] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter Octree Textures on the GPU, pages 595–613. Addison Wesley, 2005.

[LJH03] Robert S. Laramee, Bruno Jobard, and Helwig Hauser. Image space based visualization of unsteady flow on surfaces. In *Proc. of IEEE Visualization 2003*, pages 131–138, 2003.

[LKHW03] Aaron E. Lefohn, Joe M. Kniss, Charles D. Hansen, and Ross T. Whitaker. Interactive deformation and visualization of level set surfaces using graphics hardware. In *Proc. of IEEE Visualization 2003*, pages 75–82, 2003.

[LKS+06] Aaron E. Lefohn, Joe M. Kniss, Robert Strzodka, Shubhabrata Sengupta, and John D. Owens. Glift: Generic, efficient, random-access gpu data structures. *ACM Transactions on Graphics*, 25(1):60–99, January 2006.

[LMK03] Wei Li, Klaus Mueller, and Arie E. Kaufman. Empty space skipping and occlusion clipping for texture-based volume rendering. In *Proc. of IEEE Visualization 2003*, pages 317–324, 2003.

[Mau00] Sean Mauch. A fast algorithm for computing the closest point and distance transform, 2000. `http://www.acm.caltech.edu/~seanm/projects/cpt`.

[Mau03] Sean Mauch. *Effcient Algorithms for Solving Static Hamilton-Jacobi Equations*. PhD thesis, California Inst. of Techn., Perdue, CA, 2003.

[Max95] Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[MB90] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6(3):153–166, 1990.

[MBC02] Jason L. Mitchell, Chris Brennan, and Drew Card. Real-time image-space outlining for non-photorealistic rendering. In *SIGGRAPH '02 Proceedings (sketch)*, August 2002.

[MBF92] Olivier Monga, Serge Benayoun, and Olivier D. Faugeras. From partial derivatives of 3-D density images to ridge lines. In Richard A. Robb, editor, *Proceedings of SPIE Visualization in Biomedical Computing '92*, volume 1808, pages 118–129, September 1992.

[MBWB02] Ken Museth, David E. Breen, Ross T. Whitaker, and Alan H. Barr. Level set surface editing operators. In *Proc. of SIGGRAPH 2002*, pages 330–338, 2002.

[MGAK03] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[MHB+00] Michael Meißner, Jian Huang, Dirk Bartz, Klaus Mueller, and Roger Crawfis. A practical evaluation of four popular volume rendering algorithms. In *Proc. of ACM Symposium on Volume Visualization*, 2000.

[Mil94] Gavin Miller. Efficient algorithms for local and global accessibility shading. In *Proc. of SIGGRAPH '94*, pages 319–326, 1994.

[MKW+04] Gerd Marmitt, Andreas Kleer, Ingo Wald, Heiko Friedrich, and Philipp Slusallek. Fast and accurate ray-voxel intersection techniques for iso-surface ray tracing. In *Proc. of Vision, Modeling, and Visualization*, pages 429–435, 2004.

[ML94] Stephen R. Marschner and Richard J. Lobb. An evaluation of reconstruction filters for volume rendering. In R. Daniel Bergeron and Arie E. Kaufman, editors, *Proceedings of Visualization '94*, pages 100–107, 1994.

[MMK+98] Torsten Möller, Klaus Mueller, Yair Kurzion, Raghu Machiraju, and Roni Yagel. Design of accurate and smooth filters for function and derivative reconstruction. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 143–151, New York, NY, USA, 1998. ACM Press.

[Mul92] James C. Mullikin. The vector distance transform in two and three dimensions. *CVGIP: Graphical Models and Image Processing*, 54(6):526–535, November 1992.

[Mur91] Shigeru Muraki. Volumetric shape description of range data using "blobby model". *SIGGRAPH Comput. Graph.*, 25(4):227–235, 1991.

[Nad00] David R. Nadeau. Volume scene graphs. In *VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 49–56, New York, NY, USA, 2000. ACM Press.

[Nie04] Gregory M. Nielson. Radial hermite operators for scattered point cloud data with normal vectors and applications to implicitizing polygon mesh surfaces for generalized csg operations and smoothing. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 203–210, Washington, DC, USA, 2004. IEEE Computer Society.

[NMHW02] Andre Neubauer, Lukas Mroz, Helwig Hauser, and Rainer Wegenkittl. Cell-based first-hit ray casting. In *VISSYM '02: Proceedings of the symposium on Data Visualisation 2002*, pages 77–ff, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[OBA+03] Yutaka Ohtake, Alexander Belyaev, Marc Alexa, Greg Turk, and Hans-Peter Seidel. Multi-level partition of unity implicits. In *ACM Trans. Graph., Proceedings of ACM SIGGRAPH 2003*, volume 22 (3), pages 463–470, New York, NY, USA, 2003. ACM Press.

[OBS03] Yutaka Ohtake, Alexander Belyaev, and Hans-Peter Seidel. A multi-scale approach to 3D scattered data interpolation with compactly supported basis functions. In *SMI '03: Proceedings of the Shape Modeling International 2003*, page 292, Washington, DC, USA, 2003. IEEE Computer Society.

[OS88] Stanley Osher and James A. Sethian. Fronts propagating with curvature-dependent speed: algorithms based on hamilton-jacobi formulations. *J. Comput. Phys.*, 79(1):12–49, 1988.

[PPL+99] Steven Parker, Michael Parker, Yarden Livnat, Peter-Pike Sloan, Charles d. Hansen, and Peter Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):238–250, 1999.

[PS05] Emmanuel Prados and Stefano Soatto. Fast marching method for generic shape from shading. In N. Paragios, O. Faugeras, T. Chan, and C. Schnoerr, editors, *Proceedings of VLSM'05 (third International Workshop on Variational, Geometric and Level Set Methods in Computer Vision)*, volume 3752 of *Lecture Notes in Computer Science*, pages 320Ű–331. Springer, oct 2005.

[PSL+98] Steven Parker, Peter Shirley, Yarden Livnat, Charles D. Hansen, and Peter-Pike Sloan. Interactive ray tracing for isosurface rendering. In *Proc. of IEEE Visualization '98*, pages 233–238, 1998.

[RE01] Penny Rheingans and David Ebert. Volume illustration: Nonphotorealistic rendering of volume models. In *Proc. of IEEE Visualization 2001*, pages 253–264, 2001.

[RG75] Lawrence R. Rabiner and Bernard Gold. *Theory and application of digital signal processing*. Prentice-Hall, Englewood, NJ, 1975.

[RGW+03] Stefan Röttger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, and Wolfgang Straßer. Smart hardware-accelerated volume rendering. In *Proc. of VisSym 2003*, pages 231–238, 2003.

[RPZ02] Liu Ren, Hanspeter Pfister, and Matthias Zwicker. Object space ewa surface splatting: A hardware accelerated approach to high quality point rendering. In *Proc. of Eurographics 02*, pages 461–470, 2002.

[RS01] Martin Rumpf and Robert Strzodka. Level set segmentation in graphics hardware. In *IEEE Int. Conf. on Image Processing, ICIP'2001*, volume 3, pages 402–405, Thessaloniki, Greece, October 2001.

[RSC87] William T. Reeves, David Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. *Computer Graphics (Proceedings of SIGGRAPH '87)*, pages 283–291, 1987.

[RSEB+00] Christof Rezk-Salama, Klaus Engel, M. Bauer, Gunther Greiner, and Thomas Ertl. Interactive volume on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 109–118, New York, NY, USA, 2000. ACM Press.

[Sam90] Hanan Samet. *The design and analysis of spatial data structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[Set96] James A. Sethian. A fast marching level set method for monotonically advancing fronts. In *Proc. Nat. Acad. Sci.*, volume 93, pages 1591–1595, 1996.

[Set98] James A. Sethian. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry*. Cambridge University Press, 1998.

[SFYC96] Raj Shekhar, Elias Fayyad, Roni Yagel, and J. Fredrick Cornhill. Octree-based decimation of marching cubes surfaces. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 335–ff., Los Alamitos, CA, 1996. IEEE Computer Society Press.

[SGS05] Carsten Stoll, Stefan Gumhold, and Hans-Peter Seidel. Visualization with stylized line primitives. In H.Rushmeier C.T.Silva, E.Groeller, editor, *Proceedings of IEEE Visualization '05*, pages –. IEEE Computer Society Press, October 2005.

[SH05] Christian Sigg and Markus Hadwiger. Fast third-order texture filtering. In Matt Pharr, editor, *GPU Gems 2*, pages 313–329. Addison Wesley, March 2005.

[SPOK95] Vladimir. V. Savchenko, Alexander A. Pasko, Oleg G. Okunev, and Tosiyasu L. Kunii. Function representation of solids reconstructed from scattered surface points and contours. Technical Report TR 94-1-032, The University of Aizu, Japan, 1995.

[SSO94] Mark Sussman, Peter Smereka, and Stanley Osher. A level set approach for computing solutions to incompressible two-phase flow. *J. Comput. Phys.*, 114(1):146–159, 1994.

[SSP+05] Oliver Staubli, Christian Sigg, Ronald Peikert, Markus Gross, and Daniel Gubler. Volume rendering of smoke propagation cfd data. In *IEEE Visualization Conference (VIS 2005)*, Minneapolis, MN, USA, October 2005. IEEE Computer Society.

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3D shapes. In *Proc. of SIGGRAPH '90*, pages 197–206, 1990.

[SW03] Jens Schneider and Rüdiger Westermann. Compression domain volume rendering. In *Proc. of IEEE Visualization 2003*, pages 293–300, 2003.

[SW04] Scott Schaefer and Joe Warren. Dual marching cubes: Primal contouring of dual grids. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference on (PG'04)*, pages 70–76, Washington, DC, USA, 2004. IEEE Computer Society.

[TL04] Rodrigo Toledo and Bruno Lévy. Extending the graphic pipeline with new gpu-accelerated primitives. Technical report, INRIA, 2004.

[TO99] Greg Turk and James F. O'Brien. Shape transformation using variational implicit functions. In *Proc. of SIGGRAPH '99*, pages 335–342, 1999.

[Tsa00] Yen-Hsi Richard Tsai. Rapid and accurate computation of the distance function using grids. Technical report, Department of Mathematics, University of California, Los Angeles, CA, 2000.

[UA04] Yury Uralsky and Anis Ahmad. Soft shadows, July 2004. NVIDIA SDK Whitepaper.

[VKKM03] Gokul Varadhan, Shankar Krishnan, Young J. Kim, and Dinesh Manocha. Feature-sensitive subdivision and isosurface reconstruction. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 14, Washington, DC, USA, 2003. IEEE Computer Society.

[WDK01] Ming Wan, Frank Dachille, and Arie E. Kaufman. Distance-field-based skeletons for virtual navigation. In Thomas Ertl, Kenneth I. Joy, and Amitabh Varshney, editors, *Proceedings of IEEE Visualization*. IEEE Computer Society, October 2001.

[WDS99] Mason Woo, Davis, and Mary Beth Sheridan. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[WE98] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 169–177, New York, NY, USA, 1998. ACM Press.

[Wen05] Holger Wendland. *Scattered data approximation*. Cambridge Monographs on Applied and Computational Mathematics 17. Cambridge University Press, 2005.

[WGG99] Brian Wyvill, Andrew Guy, and Eric Galin. Extending the csg tree - warping, blending and boolean operations in an implicit surface modeling system. *Comput. Graph. Forum*, 18(2):149–158, 1999.

[WH94] Andrew P. Witkin and Paul S. Heckbert. Using particles to sample and control implicit surfaces. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 269–277, New York, NY, USA, 1994. ACM Press.

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

[Whi98] Ross T. Whitaker. A level-set approach to 3D reconstruction from range data. *Int. J. Comput. Vision*, 29(3):203–231, 1998.

[Wij03] Jarke J. Van Wijk. Image based flow visualization for curved surfaces. In *Proc. of IEEE Visualization 2003*, pages 745 – 754, 2003.

[WK95] Sidney W. Wang and Arie E. Kaufman. Volume sculpting. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 151–ff., New York, NY, USA, 1995. ACM Press.

[WMW86] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, 1986.

[WS01] Rüdiger Westermann and Bernd Sevenich. Accelerated volume ray-casting using texture mapping. In *Proc. of IEEE Visualization 2001*, pages 271–278, 2001.

[WWH+00] Manfred Weiler, Rüdiger Westermann, Charles D. Hansen, Kurt Zimmerman, and Thomas Ertl. Level-of-detail volume rendering via 3D textures. In *Proc. of IEEE VolVis 2000*, pages 7–13, 2000.

[ZO02] Hongkai Zhao and Stanley Osher. Visualization, analysis and shape reconstruction of unorganized data sets. In S. Osher and N. Paragios, editors, *Geometric Level Set Methods in Imaging, Vision and Graphics*. Springer-Verlag, 2002.

[ZRB+04] Matthias Zwicker, Jussi Räsänen, Mario Botsch, Carsten Dachsbacher, and Mark Pauly. Perspective accurate splatting. In *GI '04: Proceedings of the 2004 conference on Graphics interface*, pages 247–254, 2004.

Appendix A

Complete Prism Covering

Let c be a convex/concave vertex of a closed and oriented triangle mesh. Using the notation of Section 4.2.3, we want to show that a small neighborhood of c is completely covered by the union P of polyhedra. This can be best seen by taking intersections with a small sphere S centered at c (see Fig. A.1). We use the overbar symbol to denote the intersection with S. It follows that $\bar{F}_i$ are great circles, and their union $\bar{F}$ is a convex spherical polygon. Also the $\bar{A}_i$ are great circles, which can be oriented consistently toward the interior of $\bar{F}$. Finally, $\bar{P}_i$ are spherical lunes, because we can assume that the diameter of S is smaller than all edges.

The conjecture is now that S is completely covered by $\bar{P}$, the union of the lunes. By the argument given in Section 4.2.3, it is sufficient to show that the northern hemisphere is covered.

Let y be a test point on S and on the convex side of the surface, i.e. an interior point of $\bar{F}$. We choose coordinates in such a way that y is the north pole of the sphere. By connecting the vertices of the spherical polygon with the north pole, we get n spherical triangles which add up to a full 2π angle at the north pole. For the i-th triangle, let γ_i be the angle at the north pole, and α_i and β_i the angles to the meridians (see Fig. A.2).

Let us now assume that the north pole lies on the left of all $\bar{A}_i$ which can be expressed as

$$\alpha_{i+1} > \beta_i \tag{A.1}$$

Convexity implies that

$$\alpha_{i+1} + \beta_i \leq \pi \tag{A.2}$$

From Eq. A.1, Eq. A.2 and $0 < \alpha_{i+1}, \beta_i < \pi$ follows that

$$0 < \frac{\sin \beta_i}{\sin \alpha_{i+1}} < 1 \tag{A.3}$$

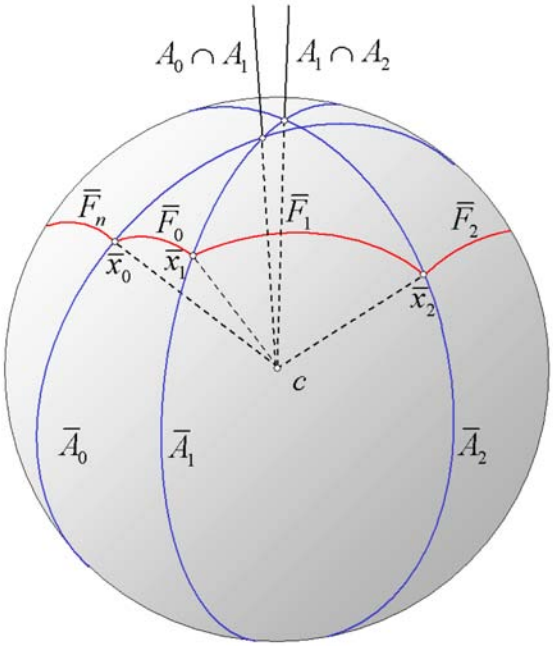


Figure A.1: Intersections of faces and angle bisector planes with the sphere.

Taking the product yields

$$\prod_{i=0}^{n-1} \frac{\sin\beta_i}{\sin\alpha_i} = \prod_{i=0}^{n-1} \frac{\sin\beta_{i+1}}{\sin\alpha_i} < 1 \tag{A.4}$$

On the other hand, we can derive

$$\prod_{i=0}^{n-1} \frac{\sin\beta_i}{\sin\alpha_i} = \prod_{i=0}^{n-1} \frac{\sin b_i}{\sin a_i} = \prod_{i=0}^{n-1} \frac{\sin b_i}{\sin b_{i+1}} = 1 \tag{A.5}$$

making use of the spherical law of sines, the fact that $a_i = b_{i+1}$ because triangles fit together, and finally $b_n = b_0$.

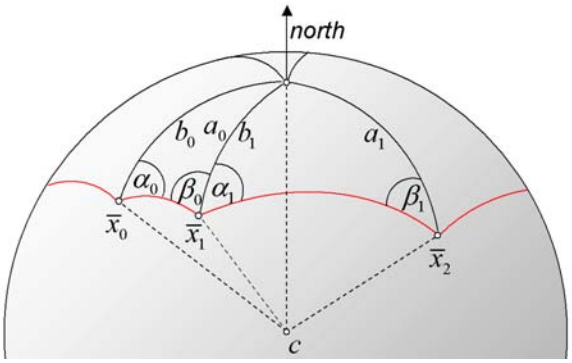


Figure A.2: Spherical triangles above mesh triangles.

From this contradiction follows that the test point is covered by $\bar{P}$, and thus also the interior of the spherical polygon $\bar{F}$ is covered.

Because of convexity, it is possible to choose an interior point of $\bar{F}$ as the north pole such that all of $\bar{F}$ lies in the northern hemisphere. It remains to show that $\bar{P}$ not only covers $\bar{F}$ but the whole hemisphere. Any spherical lune must have one of its end points below the equator, and because of convexity, this is the one on the concave side. But this means that along the equator, the sequence of lunes $\{\bar{P}_0, \cdots, \bar{P}_{n-1}\}$, $\bar{P}_0$ can't have any gaps, and therefore the hemisphere is completely covered, which was the conjecture.

Appendix B

Quadric Shader Programs

Listing B.1 and B.2 implement the vertex and fragment program for sphere rendering in OpenGL ARB program assembler. Definition of input and output alias as well as environment and temporary variables have been omited. The following coding styles and rules have been adapted for better readibility:

- Input (output) alias are marked with leading (trailing) underscores.
- Transformation matrices and their compounds are written in uppercase.
- `cn` and `rn` denote the n-th column and row of a matrix, respectively.
- `const` is a vector with components $[0, 0.5, 1, 2]$.

The vertex program first computes the coefficients of the two quadratic equations 7.9 in x and y direction. From these, the vertex position and point size in clip coordinates can be extracted. The point size needs to be specified in pixels and is thus scaled by the viewport size. The remainder of the program computes the inverse of the compound transformation matrix and the constant term in the quadratic ray equation 7.13, which is then passed to the fragment program. Computing these values once in the vertex program per quadric is faster than recomputing them in the fragment program for every pixel.

In the fragment program, the view vector in eye space is first computed from the window coordinates of the fragment. Applying the inverse transformation matrix yields the view vector in parameter space, from which the coefficients of the normalized quadratic equation for the ray intersection can be computed. If the discriminant is negative, the current pixel is not covered by the quadric and is culled. In a last step, the surface normal is transformed to eye space and written to the geometry buffer together with the fragment depth and color for deferred shading.

```
# screen planes in parameter space
MUL PMVT_c0.xyz, MVP_r0, _radius.x;
DPH PMVT_c0.w, _center, MVP_r0;

MUL PMVT_c1.xyz, MVP_r1, _radius.x;
DPH PMVT_c1.w, _center, MVP_r1;

MUL PMVT_c3.xyz, MVP_r3, _radius.x;
DPH PMVT_c3.w, _center, MVP_r3;

# parameter matrix: diag = {1,1,1,-1}
MUL PMVTD_c0, diag, PMVT_c0;
MUL PMVTD_c1, diag, PMVT_c1;
MUL PMVTD_c3, diag, PMVT_c3;

# solve two quadratic equations (x,y)
DP4 eqn.x, PMVTD_c3, PMVT_c0; # -b_x/2
DP4 eqn.z, PMVTD_c0, PMVT_c0; # c_x

DP4 eqn.y, PMVTD_c3, PMVT_c1; # -b_y/2
DP4 eqn.w, PMVTD_c1, PMVT_c1; # c_y

DP4 tmp.w, PMVTD_c3, PMVT_c3;
RCP tmp.w, tmp.w;
MUL eqn, eqn, tmp.w;

# transformed vertex position
MOV vpos_.xy, eqn;
MOV vpos_.zw, const.xxxz;

MOV result.color, vertex.color;

# radius (avoid division by zero)
MADC radius.xy, eqn, eqn, -eqn.zwxy;
RSQ tmp.x, radius.x;
RSQ tmp.y, radius.y;
MUL radius.xy (GT), radius, tmp;

# pointsize
MUL radius.xy, radius, viewport;
MAX result.pointsize.x, radius.x, radius.y;

# output inverse transformation
RCP radius.w, _radius.x;
MUL MVT_inv_c0_, MV_inv[0], radius.w;
MUL MVT_inv_c1_, MV_inv[1], radius.w;
MUL MVT_inv_c2_, MV_inv[2], radius.w;

ADD MVT_inv_c3.xyz, MV_inv[3], -_center;
MUL MVT_inv_c3.xyz, MVT_inv_c3, radius.w;
MOV MVT_inv_c3_.xyz, MVT_inv_c3;
MOV MVT_inv_c3_.w, const.z;

# output delta_p
MUL PMVT_inv_c2.xyz, MVT_inv_c3, P_inv_c2.w;
MOV PMVT_inv_c2.w, P_inv_c2.w;
MOV PMVT_inv_c2_, PMVT_inv_c2;

# output diag/a
MUL PMVTD_inv_c2, diag, PMVT_inv_c2;
DP4 tmp.w, PMVTD_inv_c2, PMVT_inv_c2;
RCP tmp.w, tmp.w;
MUL diag_div_a_, tmp.w, diag;
```

Listing B.1: Vertex program projecting the quadric to screen space and computing the point sprite parameters.

```
# inverse viewport transformation
DIV view_c.xy, fragment.position, viewport;
MAD view_c.xy, view_c, const.w, -const.z;

# view direction in eye space
MAD view_e, P_inv[1], view_c.y, P_inv[3];
MAD view_e, P_inv[0], view_c.x, view_e;

# view direction in parameter space
MUL view_p, _MVT_inv_c0, view_e.x;
MAD view_p, _MVT_inv_c1, view_e.y, view_p;
MAD view_p, _MVT_inv_c2, view_e.z, view_p;
MAD view_p, _MVT_inv_c3, view_e.w, view_p;

# quadratic equation
MUL tmp, _diag_div_a, view_p;
DP4 eqn.y, tmp, _PMVT_inv_c2;
DP4 eqn.z, tmp, view_p;
MAD eqn.w, eqn.y, eqn.y, -eqn.z;

# discriminant < 0 => kill
KIL eqn.w;

# solve quadratic equation
RSQ tmp.w, eqn.w;
MAD eqn.w, eqn.w, -tmp.w, -eqn.y;

# tranform normal to eye space
MAD view_p.xyz, _PMVT_inv_c2, eqn.w, view_p;
DP3 nrm.x, view_p, _MVT_inv_c0;
DP3 nrm.y, view_p, _MVT_inv_c1;
DP3 nrm.z, view_p, _MVT_inv_c2;
NRM nrm.xyz, nrm;

# output depth, color and normal
MAD depth_.z, eqn.w, const.y, const.y;
MOV color_, fragment.color;
MAD normal_, nrm, const.y, const.y;
```

Listing B.2: Fragment program computing the ray-quadric intersection depth and surface normal.

Copyrights

Several figures published in this thesis reproduce data sets courtesy of the following companies and institutions:

- The bunny model used in Figures 4.7, 4.10 and 6.14 is copyright of the Stanford Computer Graphics Group.
- The horse model used in Figures 3.1 and 6.13 is copyright of Cyberware Inc.
- The David model used in Figures 6.2, 6.3 and 6.12 are copyright of the Digital Michelangelo Project and the Soprintendenza ai beni artistici e storici per le province di Firenze, Pistoia, e Prato.
- The asian dragon model used in Figures 6.2, 6.8 and 6.11 is copyright of XYZ RGB Inc.
- The dragon model used in Figures 6.9 and 6.14 is copyright of the Stanford Computer Graphics Group.
- The cube data set and the color mapping functions used in Figure 6.10 are courtesy of Gordon Kindlmann.
- The medical data set used in Figures 6.2 and 6.16 are courtesy of Tiani MedGraph.
- The molecules used in Figures 7.1, 7.4, 7.5, 7.7 and 7.8 are courtesy of wwPDB.

Curriculum Vitae

Address

Name	Christian Sigg
E-Mail	sigg@inf.ethz.ch
Office	Computer Graphics Laboratory ETH Zentrum, IFW D28.2 Haldeneggsteig 4 CH-8092 Zurich, Switzerland ++41 44 632 74 76
Home	Hardturmstrasse 104 CH - 8005 Zurich, Switzerland ++41 43 300 39 51

Personal Information

Date of Birth	March 4, 1977
Nationality	Swiss
Civil Status	Married

Education

2002 - 2006	ETH Zurich, Department of Computer Science PhD in Computer Graphics, Computer Graphics Laboratory Funded by Schlumberger Research, Cambridge UK
2000 - 2001	University of Texas Austin, TX, Computer Science Department Semester Abroad at Center for Computational Visualization
1999 - 2002	ETH Zurich, Department of Mathematics Graduate Studies in Computational Science and Engineering
1997 - 1999	ETH Zurich, Department of Mathematics Undergraduate Studies in Mathematics

Professional Experience

2003 - 2006	ETH Zurich, Computer Graphics Laboratory, System Admin
1999 - 2005	ETH Zurich, Computer Graphics Laboratory, Teaching Assistant
2003	Schlumberger Research Cambridge, Summer Internship
2001 - 2002	Zurich Insurance, Zurich, Software Engineer (50%)
2000 - 2001	University of Texas Austin, TX, Student Researcher
1999 - 2000	MySign AG Aarau, Software Engineer (30%)
1997 - 1998	IBM Switzerland, Zurich, IC-Technician (30-100%)

Publications

Markus Hadwiger, Andrea Kratz, Christian Sigg, Katja Bühler, *GPU-Accelerated Deep Shadow Maps for Direct Volume Rendering*, Proceedings of ACM Siggraph/Eurographics Conference on Graphics Hardware, 2006.

Christian Sigg, Tim Weyrich, Mario Botsch, Markus Gross, *GPU-Based Ray-Casting of Quadratic Surfaces*, Proceedings of Eurographics Symposium on Point-Based Graphics, 2006.

Ovidio Mallo, Ronald Peikert, Christian Sigg, Filip Sadlo, *Illuminated Lines Revisited*, Proceedings of IEEE Visualization, 2005.

Oliver Staubli, Christian Sigg, Ronnald Peikert, Daniel Gubler, *Volume rendering of smoke propagation CFD data*Proceedings of IEEE Visualization, 2005.

Markus Hadwiger, Christian Sigg, Henning Scharsach, Katja Bühler, Markus Gross: *Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces*, Proceedings of Eurographics, 2005.

Christian Sigg, Markus Hadwiger: *Fast Third-Order Texture Filtering*, GPU Gems 2, 2005.

Ronny Peikert, Christian Sigg: *Optimized Bounding Polyhedra For GPU-Based Distance Transform*, Proceedings of Dagstuhl Seminar 023231 on Scientific Visualization, 2005.

Christian Sigg, Ronny Peikert, Markus Gross: *Signed Distance Transform using Graphics Hardware*, Proceedings of IEEE Visualization, 2003.

Christian Sigg, *Framework for Levelset Methods*, Diploma thesis, Computer Graphics Laboratory, ETH Zurich and Schlumberger Research, Cambridge, 2002.

Christian Sigg, *Parallel Ray Casting and Isocontouring of Very Large Volume Data*, Semester thesis, Center for Computational Visualization, University of Austin, TX, 2001.

Christian Sigg, *Eigenmeshes for Fairing and Resampling Problems*, Semester thesis, Computer Graphics Laboratory, ETH Zurich, 2000.